Chapter 7

Command Line

Linux system administrators spend a significant amount of their time at a command line prompt. They often automate and troubleshoot tasks in this text environment. There is a saying, "*graphical user interfaces make easy tasks easier, while command line interfaces make difficult tasks possible*". Linux relies heavily on the abundance of command line tools. The command line interface provides the following advantages:

- No GUI overhead is incurred.
- Virtually any and every task can be accomplished while sitting at the command line.
- You can implement scripts for often-used (or easy-to-forget) tasks and series of procedures.
- You can sign into remote machines anywhere on the Internet.
- You can initiate graphical applications directly from the command line instead of hunting through menus.
- While graphical tools may vary among Linux distributions, the command line interface does not.

Most input lines entered at the shell prompt have three basic elements:

- Command
- Options
- Arguments

The command is the name of the program you are executing. It may be followed by one or more options (or switches) that modify what the command may do. Options usually start with one or two dashes, for example, **-p** or **--print**, in order to differentiate them from arguments, which represent what the command operates on.

However, plenty of commands have no options, no arguments, or neither. In addition, other elements (such as setting environment variables) can also appear on the command line when launching a task.

Using a text terminal on a graphical desktop

A **terminal emulator** program emulates (simulates) a standalone terminal within a window on the desktop. By this, we mean it behaves essentially as if you were logging into the machine at a pure text terminal with no running graphical interface. Most terminal emulator programs support multiple terminal sessions by opening additional tabs or windows.

By default, on GNOME desktop environments, the **gnome-terminal** application is used to emulate a text-mode terminal in a window. Other available terminal programs include:

- **xterm**
- **rxvt**
- **konsole** (default on **KDE**)
- **terminator**

Launching Terminal Windows

To open a terminal on any system using a recent GNOME desktop click on **Applications > System Tools > Terminal** or **Applications > Utilities > Terminal**. If you do not have the **Applications** menu, you will have to install the appropriate **gnome-shell-extension** package and turn on with **gnome-tweaks**.

On any but some of the most recent GNOME-based distributions, you can always open a terminal by right-clicking anywhere on the desktop background and selecting **Open in Terminal**. If this does not work you will once again need to install and activate the appropriate **gnome-shell-extension** package.

You can also hit **Alt-F2** and type in either **gnome-terminal** or **konsole**, whichever is appropriate.

Because distributions have had a history of burying opening up a command line terminal, and the place in menus may vary in the desktop GUI, It is a good idea to figure out how to "pin" the terminal icon to the panel, which might mean adding it to the Favorites grouping on GNOME systems.

Some basic utilities

- **cat**: used to type out a file (or combine files)
- **head**: used to show the first few lines of a file
- **tail**: used to show the last few lines of a file
- **man**: used to view documentation.

sudo command

All the demonstrations created have a user configured with **sudo** capabilities to provide the user with administrative (admin) privileges when required. **sudo** allows users to run programs using the security privileges of another user, generally root (superuser).

Steps for setting up and running sudo

1. You will need to make modifications as the administrative or superuser, root. While **sudo** will become the preferred method of doing this, we do not have it set up yet, so we will use **su** (which we will discuss later in detail) instead. At the command line prompt, type **su** and press **Enter**. You will then be prompted for the root password, so enter it and press **Enter**. You will notice that nothing is printed; this is so others cannot see the password on the screen. You should end up with a different looking prompt, often ending with '#'. For example:
   ```
   $ su Password:
   #
   ```
2. Now, you need to create a configuration file to enable your user account to use sudo. Typically, this file is created in the **/etc/sudoers.d/** directory with the name of the file the same as your username. For example, for this demo, let's say your username is **student**. After doing step 1, you would then create the configuration file for **student** by doing this:
   ```
   # echo "student ALL=(ALL) ALL" > /etc/sudoers.d/student
   ```
3. Finally, some Linux distributions will complain if you do not also change permissions on the file by doing:
   ```
   # chmod 440 /etc/sudoers.d/student
   ```

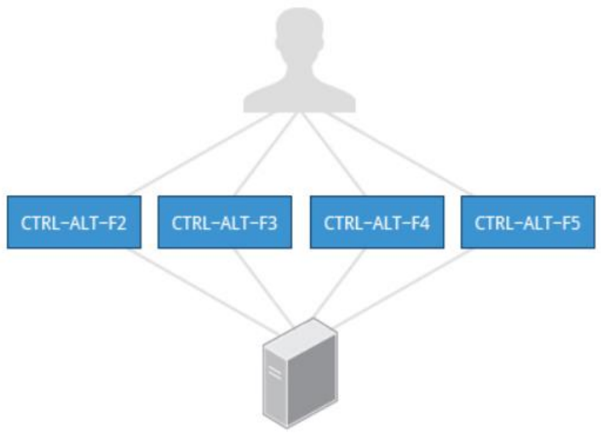
Virtual Terminals

Virtual Terminals (**VT**) are console sessions that use the entire display and keyboard outside of a graphical environment. Such terminals are considered "virtual" because, although there can be multiple active terminals, only one terminal remains visible at a time. A VT is not quite the same as a command line terminal window; you can have many of those visible at once on a graphical desktop.

One virtual terminal (usually number one or seven) is reserved for the graphical environment, and text logins are enabled on the unused VTs. Ubuntu uses VT 7, but CentOS/RHEL and openSUSE use VT 1 for the graphical display.

An example of a situation where using VTs is helpful is when you run into problems with the graphical desktop. In this situation, you can switch to one of the text VTs and troubleshoot.

To switch between VTs, press **CTRL-ALT-function key** for the VT. For example, press **CTRL-ALT-F6** for VT 6. Actually, you only have to press the **ALT-F6** key combination if you are in a VT and want to switch to another VT.



Turning off the graphical desktop

or the newer systemd-based distributions, the display manager is run as a service, you can stop the GUI desktop with the systemctl utility and most distributions will also work with the **telinit** command, as in:

$ sudo systemctl stop gdm (or **sudo telinit 3**)

and restart it (after logging into the console) with:

$ sudo systemctl start gdm (or **sudo telinit 5**)

On Ubuntu versions before 18.04 LTS, substitute **lightdm** for **gdm**.

Basic Operations

1. **Reboot and shutdown**

The preferred method to shut down or reboot the system is to use the **shutdown** command. The **halt** and **poweroff** commands issue **shutdown -h** to halt the system; **reboot** issues **shutdown -r** and causes the machine to reboot instead of just shutting down.

2. **Locating applications**

In general, executable programs and scripts should live in the **/bin**, **/usr/bin**, **/sbin**, **/usr/sbin** directories, or somewhere under **/opt**. They can also appear in **/usr/local/bin** and **/usr/local/sbin**, or in a directory in a user's account space, such as **/home/student/bin**.

One way to locate programs is to employ the **which** utility.

Syntax: which <file-name>

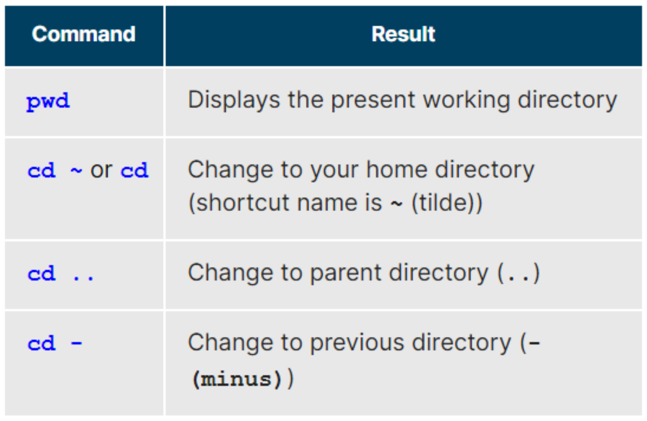
Eg: which diff

Another alternative is whereis. it looks for packages in a broader range of system directories.

Eg: whereis diff

3. **Accessing directories**

When you first log into a system or open a terminal, the default directory should be your home directory. You can print the exact path of this by typing **echo $HOME**. For directory navigation:



Command	Result
`pwd`	Displays the present working directory
`cd ~` or `cd`	Change to your home directory (shortcut name is ~ (tilde))
`cd ..`	Change to parent directory (`..`)
`cd -`	Change to previous directory (`-` `(minus)`)

4. **Absolute and Relative Paths**

There are two ways to identify paths:

- **Absolute pathname**
  An absolute pathname begins with the root directory and follows the tree, branch by branch, until it reaches the desired directory or file. Absolute paths always start with **/**.
- **Relative pathname**
  A relative pathname starts from the present working directory. Relative paths never start with **/**.

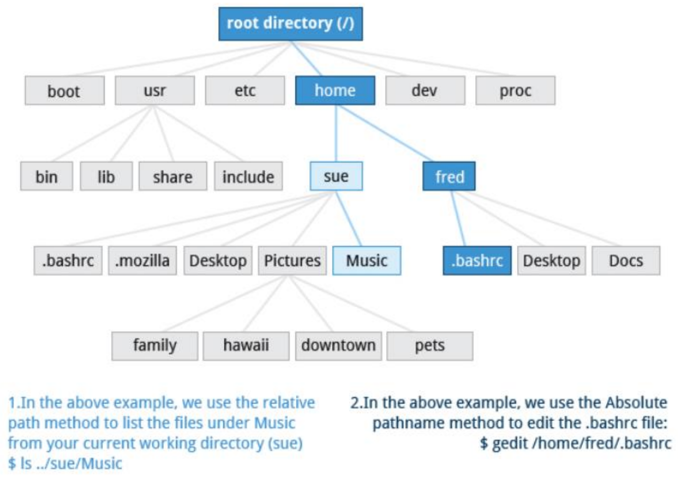
Multiple slashes (**/**) between directories and files are allowed, but all but one slash between elements in the pathname is ignored by the system. **////usr//bin** is valid, but seen as **/usr/bin** by the system.

Most of the time, it is most convenient to use relative paths, which require less typing. Usually, you take advantage of the shortcuts provided by: **.** (present directory), **..** (parent directory) and **~** (your home directory).

For example, suppose you are currently working in your home directory and wish to move to the **/usr/bin** directory. The following two ways will bring you to the same directory from your home directory:

- Absolute pathname method
  $ cd /usr/bin
- Relative pathname method
  $ cd ../../usr/bin

In this case, the absolute pathname method requires less typing.


root directory (/)
boot
usr
etc
home
dev
proc
bin
lib
share
include
sue
fred
.bashrc
.mozilla
Desktop
Pictures
Music
.bashrc
Desktop
Docs
family
hawaii
downtown
pets
1.In the above example, we use the relative path method to list the files under Music from your current working directory (sue)
$ ls ../sue/Music
2.In the above example, we use the Absolute pathname method to edit the .bashrc file:
$ gedit /home/fred/.bashrc

5. **Tree command**

The **tree** command is a good way to get a bird's-eye view of the filesystem tree.
Use **tree -d** to view just the directories and to suppress listing file names. Following commands can be used to explore the file system:

Command	Usage
`cd /`	Changes your current directory to the root (/) directory (or path you supply)
`ls`	List the contents of the present working directory
`ls –a`	List all files, including hidden files and directories (those whose name start with .)
`tree`	Displays a tree view of the filesystem

The **cd** command remembers where you were last, and lets you get back there with **cd –**
The list of directories is displayed with the **dirs** command.
To push the current working directory into the stack, type **pushd .** (fullstop included)
To pop the topmost directory from stack press **popd**

Hard links

The **ln** utility is used to create hard links and (with the **-s** option) soft links, also known as symbolic links or symlinks. These two kinds of links are very useful in UNIX-based operating systems.

Suppose that **file1** already exists. A hard link, called **file2**, is created with the command:

```
$ ln file1 file2
```

Note that two files now appear to exist. However, a closer inspection of the file listing shows that this is not quite true.

```
$ ls -li file1 file2
```

The **-i** option to **ls** prints out in the first column the inode number, which is a unique quantity for each file object. This field is the same for both of these files; what is really going on here is that it is only one, file but it has more than one name associated with it, as is indicated by the **2** that appears in the **ls** output. Thus, there was already another object linked to **file1** before the command was executed.

Hard links are very useful and they save space, but you have to be careful with their use, sometimes in subtle ways. For one thing, if you remove either **file1** or **file2** in the example, the inode object (and the remaining file name) will remain, which might be undesirable, as it may lead to subtle errors later if you recreate a file of that name.

If you edit one of the files, exactly what happens depends on your editor; most editors, including **vi** and **gedit**, will retain the link *by default*, but it is possible that modifying one of the names may break the link and result in the creation of two objects.

Soft Links

Soft (or Symbolic) links are created with the **-s** option, as in:

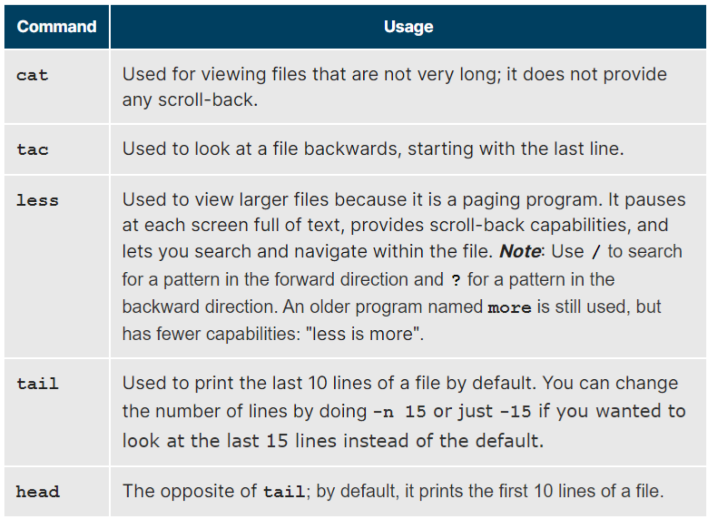
```
$ ln -s file1 file3
$ ls -li file1 file3
```

Notice **file3** no longer appears to be a regular file, and it clearly points to **file1** and has a different inode number.

Symbolic links take no extra space on the filesystem (unless their names are very long). They are extremely convenient, as they can easily be modified to point to different places. An easy way to create a shortcut from your **home** directory to long pathnames is to create a symbolic link.

Unlike hard links, soft links can point to objects even on different filesystems, partitions, and/or disks and other media, which may or may not be currently available or even exist. In the case where the link does not point to a currently available or existing object, you obtain a dangling link.

Viewing files



Command	Usage
`cat`	Used for viewing files that are not very long; it does not provide any scroll-back.
`tac`	Used to look at a file backwards, starting with the last line.
`less`	Used to view larger files because it is a paging program. It pauses at each screen full of text, provides scroll-back capabilities, and lets you search and navigate within the file. ***Note***: Use `/` to search for a pattern in the forward direction and `?` for a pattern in the backward direction. An older program named `more` is still used, but has fewer capabilities: "less is more".
`tail`	Used to print the last 10 lines of a file by default. You can change the number of lines by doing `-n 15` or just `-15` if you wanted to look at the last 15 lines instead of the default.
`head`	The opposite of `tail`; by default, it prints the first 10 lines of a file.

touch command

touch is often used to set or update the access, change, and modify times of files. By default, it resets a file's timestamp to match the current time. Can also be used to create an empty file

Syntax: touch <file-name>

the **-t** option allows you to set the date and timestamp of the file to a specific value, as in:

```
$ touch -t 12091600 myfile
```

This sets the **myfile** file's timestamp to 4 p.m., December 9th (12 09 1600).

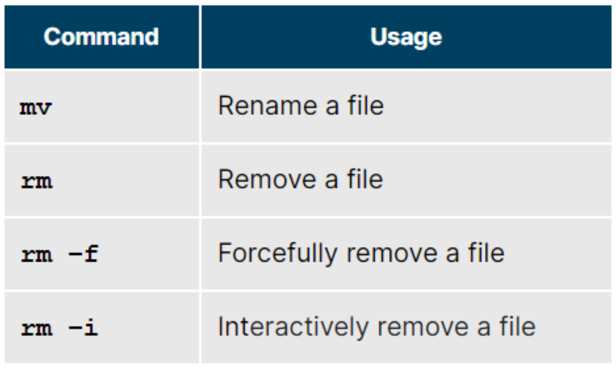
mkdir and rmdir

the **-t** option allows you to set the date and timestamp of the file to a specific value, as in:

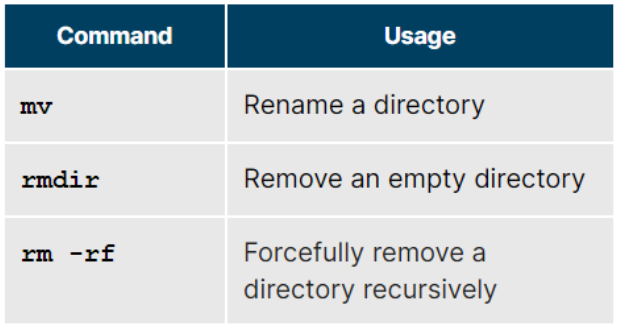
```
$ touch -t 12091600 myfile
```

This sets the **myfile** file's timestamp to 4 p.m., December 9th (12 09 1600).

Removing a directory is done with **rmdir.** The directory must be empty or the command will fail. To remove a directory and all of its contents you have to do **rm -rf**.



Command	Usage
`mv`	Rename a file
`rm`	Remove a file
`rm –f`	Forcefully remove a file
`rm –i`	Interactively remove a file



Command	Usage
`mv`	Rename a directory
`rmdir`	Remove an empty directory
`rm -rf`	Forcefully remove a directory recursively

Modifying the command line prompt

The **PS1** variable is the character string that is displayed as the prompt on the command line. Most distributions set **PS1** to a known default value, which is suitable in most cases. However, users may want custom information to show on the command line. For example, some system administrators require the user and the host system name to show up on the command line as in:

```
student@c8 $
```

This could prove useful if you are working in multiple roles and want to be always reminded of who you are and what machine you are on. The prompt above could be implemented by setting the **PS1** variable to: **\u@\h \$**.

For example:

```
$ echo $PS1
\$
```

```
$ PS1="\u@\h \$ "
student@c8 $ echo $PS1
\u@\h \$
student@c8 $
```

By convention, most systems are set up so that the root user has a pound sign (#) as their prompt.

Standard File Streams

When commands are executed, by default there are three standard file streams (or descriptors) always open for use: standard input (standard in or **stdin**), standard output (standard out or **stdout**) and standard error (or **stderr**).

Name	Symbolic Name	Value	Example
standard input	**stdin**	0	keyboard
standard output	**stdout**	1	terminal
standard error	**stderr**	2	log file

I/O redirection

Through the command shell**,** we can redirect the three standard file streams so that we can get input from either a file or another command, instead of from our keyboard, and we can write output and errors to files or use them to provide input for subsequent commands.

For example, if we have a program called **do_something** that reads from **stdin** and writes to **stdout** and **stderr**, we can change its input source by using the less-than sign (**<**) followed by the name of the file to be consumed for input data:

```
$ do_something < input-file
```

If you want to send the output to a file, use the greater-than sign (**>**) as in:

```
$ do_something > output-file
```

Because **stderr** is not the same as **stdout**, error messages will still be seen on the terminal windows in the above example.

If you want to redirect **stderr** to a separate file, you use **stderr**'s file descriptor number (2), the greater-than sign (**>**), followed by the name of the file you want to hold everything the running command writes to **stderr**:

```
$ do_something 2> error-file
```

Note: By the same logic, **do_something 1> output-file** is the same as **do_something > output-file**.

A special shorthand notation can send anything written to file descriptor **2** (**stderr**) to the same place as file descriptor **1** (**stdout**): **2>&1**.

```
$ do_something > all-output-file 2>&1
```

bash permits an easier syntax for the above:

```
$ do_something >& all-output-file
```

Pipes

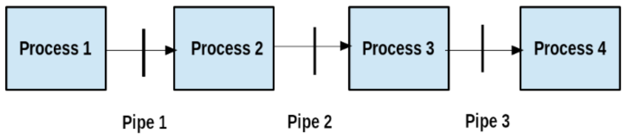
The UNIX/Linux philosophy is to have many simple and short programs (or commands) cooperate together to produce quite complex results, rather than have one complex program with many possible options and modes of operation. In order to accomplish this, extensive use of pipes is made. You can pipe the output of one command or program into another as its input.

In order to do this, we use the vertical-bar, **|**, (pipe symbol) between commands as in:

```
$ command1 | command2 | command3
```

The above represents what we often call a pipeline, and allows Linux to combine the actions of several commands into one. This is extraordinarily efficient because **command2** and **command3** do not have to wait for the previous pipeline commands to complete before they can begin hacking at the data in their input streams; on multiple CPU or core systems, the available computing power is much better utilized and things get done quicker.

Furthermore, there is no need to save output in (temporary) files between the stages in the pipeline, which saves disk space and reduces reading and writing from disk, which is often the slowest bottleneck in getting something done.



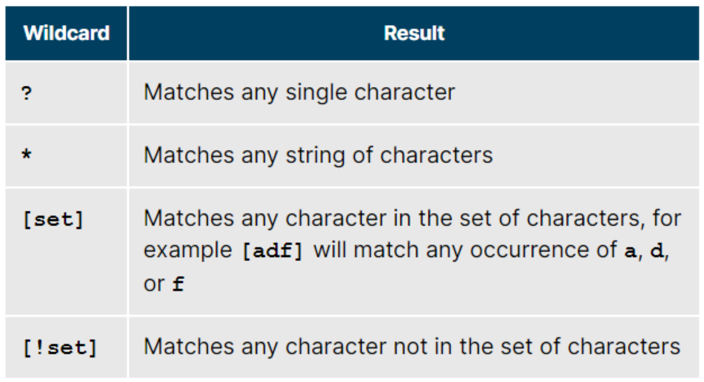
Pipeline

Searching for files

1. **Locate**
2. The **locate** utility program performs a search taking advantage of a previously constructed database of files and directories on your system, matching all entries that contain a specified character string. This can sometimes result in a very long list.
3. To get a shorter (and possibly more relevant) list, we can use the **grep** program as a filter. **grep** will print only the lines that contain one or more specified strings, as in:
4. **$ locate zip | grep bin**
5. which will list all the files and directories with both **zip** and **bin** in their name. We will cover **grep** in much more detail later. Notice the use of **|** to pipe the two commands together.
6. **locate** utilizes a database created by a related utility, **updatedb**. Most Linux systems run this automatically once a day. However, you can update it at any time by just running **updatedb** from the command line as the root user.

2. **Wildcards**

You can search for a filename containing specific characters using wildcards.



Wildcard	Result
`?`	Matches any single character
`*`	Matches any string of characters
`[set]`	Matches any character in the set of characters, for example `[adf]` will match any occurrence of `a`, `d`, or `f`
`[!set]`	Matches any character not in the set of characters

To search for files using the **?** wildcard, replace each unknown character with **?**. For example, if you know only the first two letters are 'ba' of a three-letter filename with an extension of **.out**, type **ls ba?.out** .

To search for files using the ***** wildcard, replace the unknown string with *****. For example, if you remember only that the extension was **.out**, type **ls *.out**.

3. The find program
4. **find** is an extremely useful and often-used utility program in the daily life of a Linux system administrator. It recurses down the filesystem tree from any particular directory (or set of directories) and locates files that match specified conditions. The default pathname is always the present working directory.
5. For example, administrators sometimes scan for potentially large core files (which contain diagnostic information after a program fails) that are more than several weeks old in order to remove them.
6. It is also common to remove files in inessential or outdated files in **/tmp** (and other volatile directories, such as those containing cached files) that have not been accessed recently. Many Linux distributions use shell scripts that run periodically (through **cron** usually) to perform such house cleaning.
7. When no arguments are given, **find** lists all files in the current directory and all of its subdirectories. Commonly used options to shorten the list include **-name** (only list files with a certain pattern in their name), **-iname** (also ignore the case of file names), and **-type** (which will restrict the results to files of a certain specified type, such as **d** for directory, **l** for symbolic link, or **f** for a regular file, etc.).
8. Searching for files and directories named **gcc**:

   ```
   $ find /usr -name gcc
   ```
9. Searching only for directories named **gcc**:

   ```
   $ find /usr -type d -name gcc
   ```
10. Searching only for regular files named **gcc**:

    ```
    $ find /usr -type f -name gcc
    ```

Advance find options

Another good use of **find** is being able to run commands on the files that match your search criteria. The **-exec** option is used for this purpose.

To find and remove all files that end with **.swp**:

```
$ find -name "*.swp" -exec rm {} ’;’
```

The **{}** (squiggly brackets) is a placeholder that will be filled with all the file names that result from the find expression, and the preceding command will be run on each one individually.

Please note that you have to end the command with either ‘**;**’ (including the single-quotes) or "**\;**". Both forms are fine.

One can also use the **-ok** option, which behaves the same as **-exec**, except that **find** will prompt you for permission before executing the command. This makes it a good way to test your results before blindly executing any potentially dangerous commands.

Finding files based on time and size

It is sometimes the case that you wish to find files according to attributes, such as when they were created, last used, etc., or based on their size. It is easy to perform such searches.

To find files based on time:

```
$ find / -ctime 3
```

Here, **-ctime** is when the inode metadata (i.e. file ownership, permissions, etc.) last changed; it is often, but not necessarily, when the file was first created. You can also search for accessed/last read (**-atime**) or modified/last written (**-mtime**) times. The number is the number of days and can be expressed as either a number (**n**) that means exactly that value, **+n**, which means greater than that number, or **-n**, which means less than that number. There are similar options for times in minutes (as in **-cmin**, **-amin**, and **-mmin**).

To find files based on sizes:

```
$ find / -size 0
```

Note the size here is in 512-byte blocks, by default; you can also specify bytes (c), kilobytes (k), megabytes (M), gigabytes (G), etc. As with the time numbers above, file sizes can also be exact numbers (**n**), **+n** or **-n**. For details, consult the man page for find.

For example, to find files greater than 10 MB in size and running a command on those files:

```
$ find / -size +10M -exec command {} ’;’
```

Package management systems on Linux

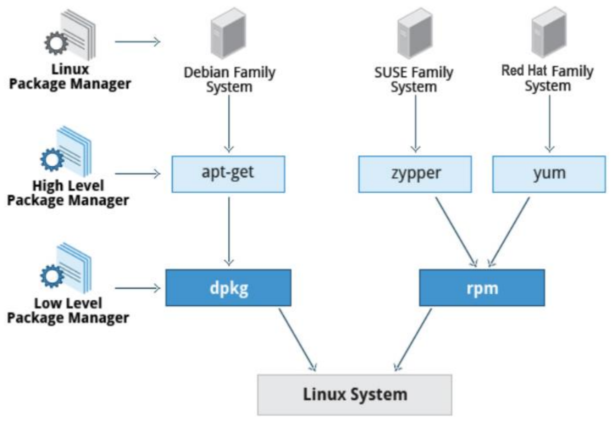
The core parts of a Linux distribution and most of its add-on software are installed via the **Package Management System**. Each package contains the files and other instructions needed to make one software component work well and cooperate with the other components that comprise the entire system. Packages can depend on each other. For example, a package for a web-based application written in **PHP** can depend on the **PHP** package.

There are two broad families of package managers: those based on **Debian** and those which use **RPM** as their low-level package manager. The two systems are incompatible, but broadly speaking, provide the same features and satisfy the same needs. There are some other systems used by more specialized Linux distributions.

Package managers: Two Levels

Both package management systems operate on two distinct levels: a low-level tool (such as **dpkg** or **rpm**) takes care of the details of unpacking individual packages, running scripts, getting the software installed correctly, while a high-level tool (such as **apt**, **yum**, **dnf** or **zypper**) works with groups of packages, downloads packages from the vendor, and figures out dependencies.

Most of the time users need to work only with the high-level tool, which will take care of calling the low-level tool as needed. Dependency resolution is a particularly important feature of the high-level tool, as it handles the details of finding and installing each dependency for you. Be careful, however, as installing a single package could result in many dozens or even hundreds of dependent packages being installed.



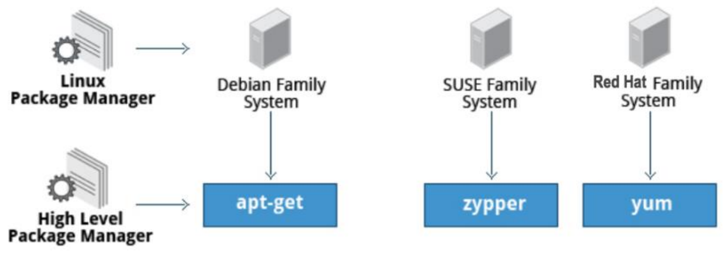
Working with different package management systems

The **A**dvanced **P**ackaging **T**ool (**apt**) is the underlying package management system that manages software on Debian-based systems. While it forms the backend for graphical package managers, such as the Ubuntu Software Center and synaptic, its native user interface is at the command line, with programs that include **apt** (or **apt-get**) and **apt-cache**.

yum is an open source command-line package-management utility for the RPM-compatible Linux systems that belongs to the Red Hat family. **yum** has both command line and graphical user interfaces. Fedora and RHEL 8 have replaced **yum** with **dnf**, which has less historical baggage, has nice new capabilities and is mostly backwards-compatible with **yum** for day-to-day commands.

zypper is the package management system for the SUSE/openSUSE family and is also based on RPM. **zypper** also allows you to manage repositories from the command line. **zypper** is fairly straightforward to use and resembles **yum** quite closely.

To learn the basic packaging commands, take a look at these basic packaging commands.



Summary

- Virtual terminals (VT) in Linux are consoles, or command line terminals that use the connected monitor and keyboard.
- Different Linux distributions start and stop the graphical desktop in different ways.
- A terminal emulator program on the graphical desktop works by emulating a terminal within a window on the desktop.
- The Linux system allows you to either log in via text terminal or remotely via the console.
- When typing your password, nothing is printed to the terminal, not even a generic symbol to indicate that you typed.
- The preferred method to shut down or reboot the system is to use the **shutdown** command.
- There are two types of pathnames**:** absolute and relative.
- An absolute pathname begins with the root directory and follows the tree, branch by branch, until it reaches the desired directory or file.
- A relative pathname starts from the present working directory.
- Using hard and soft (symbolic) links is extremely useful in Linux.
- **cd** remembers where you were last, and lets you get back there with **cd -**.
- **locate** performs a database search to find all file names that match a given pattern.
- **find** locates files recursively from a given directory or set of directories.
- **find** is able to run commands on the files that it lists, when used with the **-exec** option.
- **touch** is used to set the access, change, and edit times of files, as well as to create empty files.
- The Advanced Packaging Tool (apt) package management system is used to manage installed software on Debian-based systems.
- You can use the Yellowdog Updater Modified (yum) open source command-line package management utility for RPM-compatible Linux operating systems.
- The zypper package management system is based on RPM and used for openSUSE.