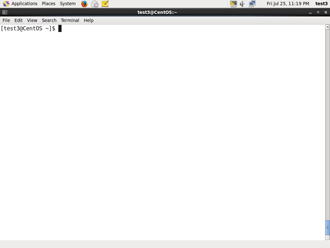
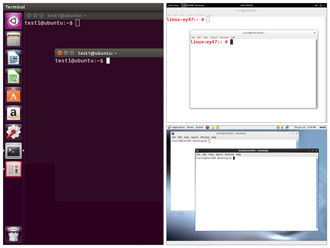
**Introduction to the Command Line**



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.
* Virtually every task can be accomplished using the command line.
* You can script tasks and series of procedures.
* You can log on remotely to networked machines anywhere on the Internet.
* You can initiate graphical apps directly from the command line.

**Using a Text Terminal on the Graphical Desktop**

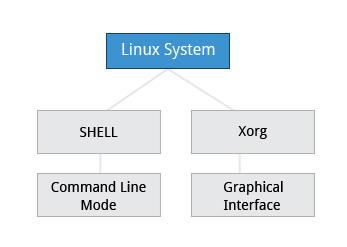


A **terminal emulator** program emulates (simulates) a stand alone terminal within a window on the desktop. By this we mean it behaves essentially as if you were logging into the machine at a pure text terminal with no running graphical interface. Most terminal emulator programs support multiple terminal 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**
* **terminator**

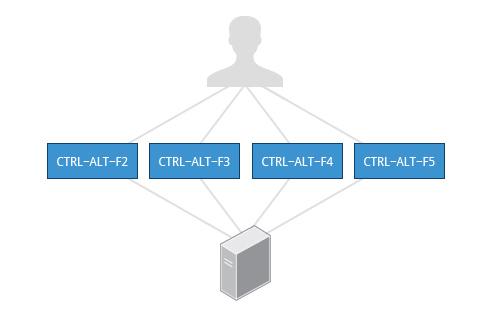
**The X Window System**



The customizable nature of Linux allows you to drop (temporarily or permanently) the **X Window** graphical interface, or to start it up after the system has been running. Certain Linux distributions distinguish versions of the install media between desktop (with **X**) and server (usually without **X**); Linux production servers are usually installed without **X** and even if it is installed, usually do not launch it during system start up. Removing **X** from a production server can be very helpful in maintaining a lean system which can be easier to support and keep secure.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)

**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 the VTs is helpful 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 the VTs, press **CTRL-ALT-corresponding function key** for the VT. For example, press **CTRL-ALT-F6** for VT 6. (Actually you only have to press **ALT-F6** key combination if you are ina VT not running **X** and want to switch to another VT.)

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)

**The Command Line**

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. You can also type other things at the command line besides issuing commands, such as setting environment variables.

**Turning off the Graphical Desktop**



Linux distributions can start and stop the graphical desktop in various ways. For **Debian**-based systems, the **Desktop Manager** runs as a service which can be simply stopped. For RPM-based systems, the **Desktop** **Manager** is run directly by **init** when set to run level 5; switching to a different runlevel stops the desktop.

Use the sudo service gdm stop or sudo service lightdm stop commands, to stop the graphical user interface in **Debian**-based systems. On **RPM**-based systems typing sudo telinit 3 may have the same effect of killing the GUI.

**sudo**

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). The functionality of **sudo** is similar to that of **run as** in **Windows**.

On your own systems, you may need to set up and enable **sudo** to work correctly. To do this, you need to follow some steps that we won’t explain in much detail now, but you will learn about later in this course. When running on **Ubuntu**, **sudo** is already always set up for you during installation. If you are running something in the **Fedora** or **openSUSE** families of distributions, you will likely need to set up **sudo** to work properly for you after initial installation.

Next, you will learn the steps to setup and run **sudo** on your system.

**Steps for Setting up and Running sudo**

If your system does not already have **sudo** set up and enabled, you need to do the following steps:

1. You will need to make modifications as the administrative or super user, root. While **sudo** will become the preferred method of doing this, we don’t 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 don’t also change permissions on the file by doing: # chmod 440 /etc/sudoers.d/student

That should be it. For the rest of this course, if you use **sudo** you should be properly set up. When using **sudo,** by default you will be prompted to give a password (your own user password) at least the first time you do it within a specificed time interval. It is possible (though very insecure) to configure **sudo** to not require a password or change the time window in which the password does not have to be repeated with every **sudo** command.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a025f50546dd4b1894c6c9a72dfb654b/1#)

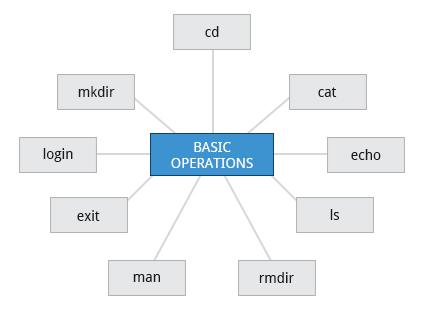
**Install GUI Ubuntu**

sudo apt-get install ubuntu-desktop

Start GUI

startx

**Basic Operations**



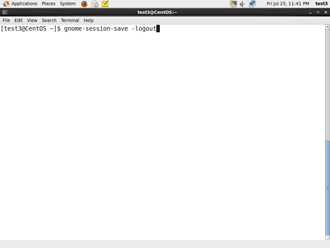
In this section we will discuss how to accomplish basic operations from the command line. These include how to log in and log out from the system, restart or shutdown the system, locate applications, access directories, identify the absolute and relative paths, and explore the filesystem.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**Logging In and Out**

An available **text terminal** will prompt for a username (with the string login:) and password. When typing your password, nothing is displayed on the terminal (not even a \* to indicate that you typed in something) to prevent others from seeing your password. After you have logged in to the system, you can perform basic operations.

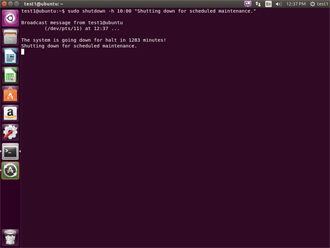
Once your session is started (either by logging in to a text terminal or via a graphical terminal program) you can also connect and log in to remote systems via the **Secure** **Shell (SSH)** utility. For example, by typing ssh username@remote-server.com, **SSH** would connect securely to the remote machine and give you a command line terminal window, using passwords (as with regular logins) or cryptographic keys (a topic we won't discuss) to prove your identity.



Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**Rebooting and Shutting Down**



The preferred method to shut down or reboot the system is to use the **shutdown** command. This sends a warning message and then prevents further users from logging in. The **init** process will then control shutting down or rebooting the system. It is important to always shut down properly; failure to do so can result in damage to the system and/or loss of data.

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. Both rebooting and shutting down from the command line requires superuser (root) access.

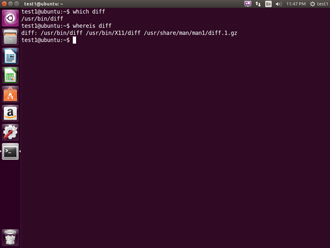
When administering a multiuser system, you have the option of notifying all users prior to shutdown as in:

$ sudo shutdown -h 10:00 "Shutting down for scheduled maintenance."

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**Locating Applications**

Depending on the specifics of your particular distribution's policy, programs and software packages can be installed in various directories. In general, executable programs should live in the /bin, /usr/bin,/sbin,/usr/sbin directories or under /opt.

One way to locate programs is to employ the **which** utility. For example, to find out exactly where the **diff** program resides on the filesystem:

$ which diff

If **which** does not find the program, **whereis** is a good alternative because it looks for packages in a broader range of system directories:

$ whereis diff

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**Accessing Directories**

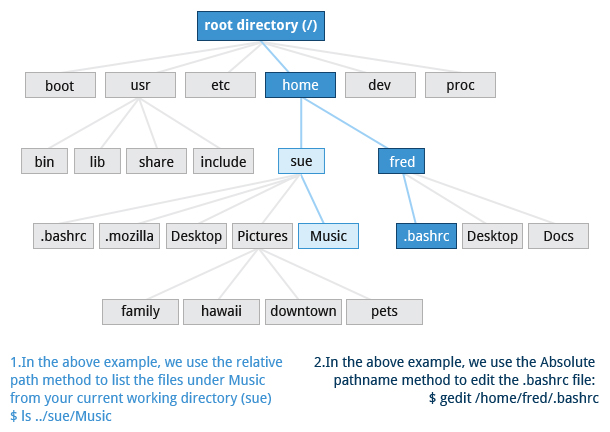
When you first log into a system or open a terminal, the default directory should be your **home directory**; you can print the exact path of this by typing echo $HOME. (Note that some Linux distributions actually open new **graphical** terminals in $HOME/Desktop.) The following commands are useful for directory navigation:

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

**Note: The next two screens cover a demonstration and Try-It-Yourself activity. You can view a demonstration and practice the Try-It-Yourself activity.**

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**Understanding Absolute and Relative Paths**



There are two ways to identify paths:

1. **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 /.
2. **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:

1. Absolute pathname method: $ cd /usr/bin
2. Relative pathname method: $ cd ../../usr/bin

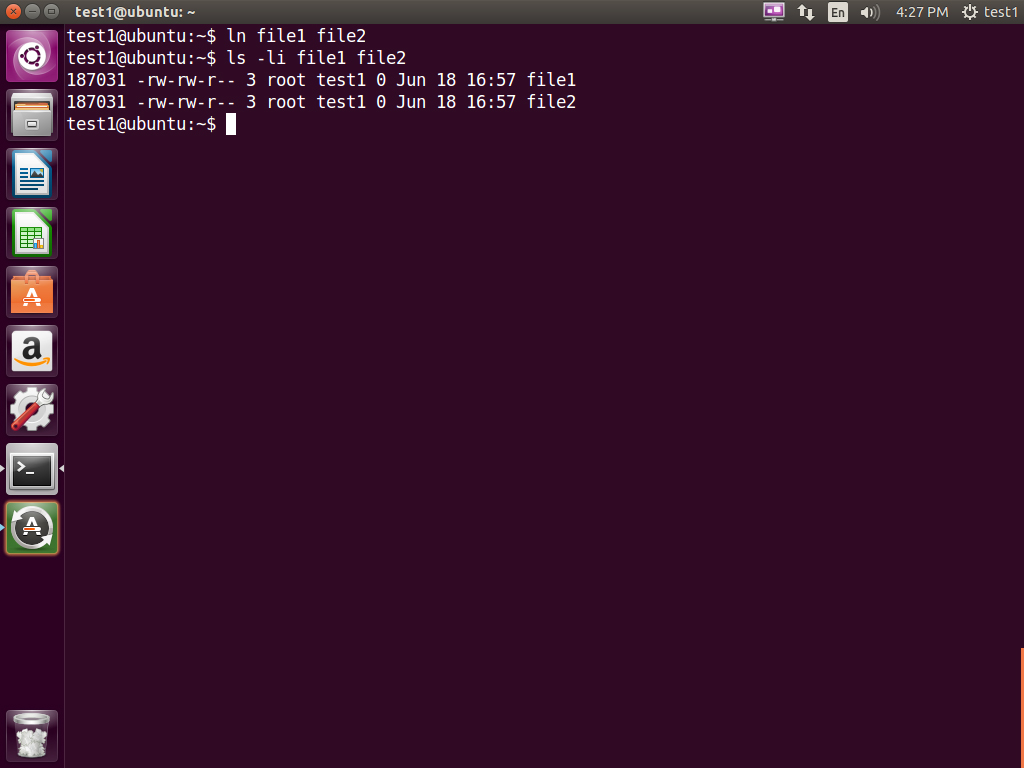
**Exploring the Filesystem**

Traversing up and down the filesystem tree can get tedious. 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.

The following commands can help in exploring the filesystem:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| cd / | **C**hanges your current **d**irectory to the root (/) directory (or path you supply) |
| ls | **L**i**s**t the contents of the present working directory |
| ls –a | **L**i**s**t **all** files including **hidden** files and directories (those whose name start with . ) |
| tree | Displays a **tree** view of the filesystem |

**Hard and Soft (Symbolic) Links**



**ln** can be 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. The advantages of symbolic links are discusssed on the following screen.

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

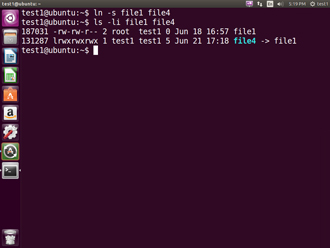
$ ln file1 file2

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 nameassociated with it, as is indicated by the **3** that appears in the **ls** output. Thus, there already was another object linked to file1 before the command was executed.

**Symbolic Links**



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

$ ln -s file1 file4

$ ls -li file1 file4

Notice file4 no longer appears to be a regular file, and it clearly points to file1 and has a different inode number.

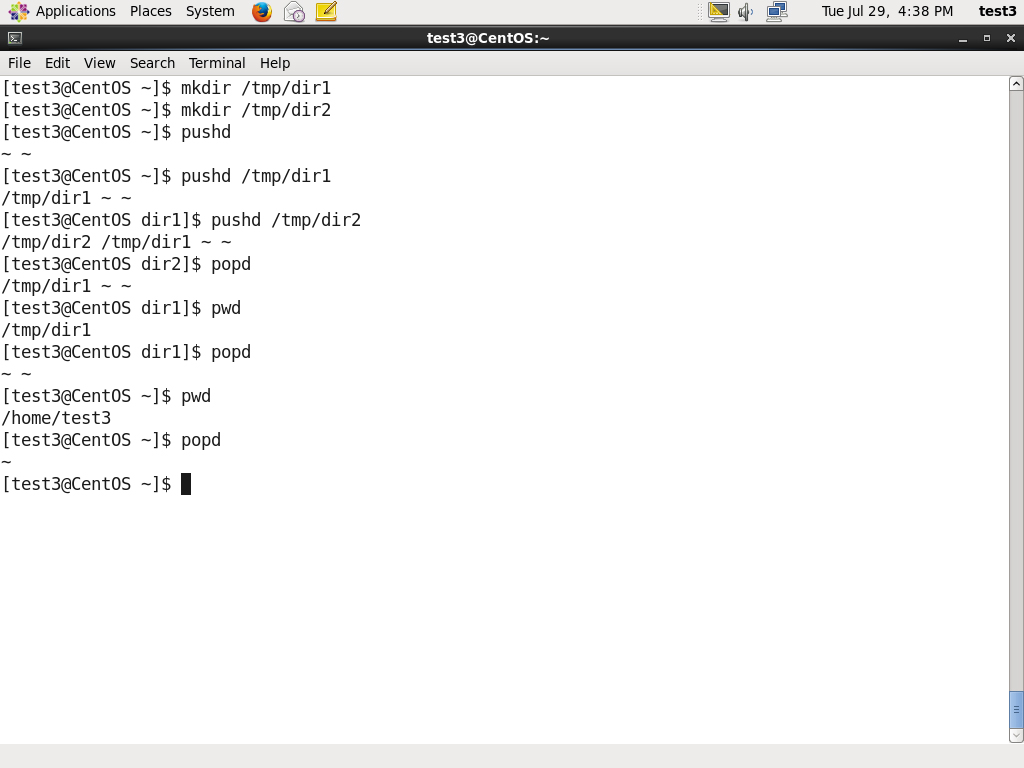
Symbolic links take no extra space on the filesystem (unless their names are very long). They are extremely convenient as they can easily be modified to point to different places. An easy way to create a shortcut from your **home** directory to long pathnames is to create a symbolic link.

Unlike hard links, soft links can point to objects even on different filesystems (or partitions) 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.

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 on the previous screen, 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 youedit 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.

**Navigating the Directory History**



The **cd** command remembers where you were last, and lets you get back there with cd -. For remembering more than just the last directory visited, use **pushd** to change the directory instead of **cd**; this pushes your starting directory onto a list. Using **popd** will then send you back to those directories, walking in reverse order (the most recent directory will be the first one retrieved with **popd**). The list of directories is displayed with the **dirs** command.

**Note: The next screen provides a video demonstration of using navigation.**

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/a3c81bd2748740c3a7bd4871f6fdfc30/1#)

**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**). Usually, **stdin** is your keyboard, **stdout** and **stderr** are printed on your terminal; often **stderr** is redirected to an error logging file. **stdin** is often supplied by directing input to come from a file or from the output of a previous command through a **pipe**. **stdout** is also often redirected into a file. Since **stderr** is where error messages are written, often nothing will go there.

In Linux, all open files are represented internally by what are called **file descriptors**. Simply put, these are represented by numbers starting at zero. **stdin** is file descriptor 0, **stdout** is file descriptor 1, and **stderr** is file descriptor 2. Typically, if other files are opened in addition to these three, which are opened by default, they will start at file descriptor 3 and increase from there.

On the next screen and in chapters ahead, you will see examples which alter where a running command gets its input, where it writes its output, or where it prints diagnostic (error) messages.

**I/O Redirection**

Through the command **shell** we can **redirect** the three standard filestreams 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 send them as 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.

**I/O Redirection**

Through the command **shell** we can **redirect** the three standard filestreams 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 send them as 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

A special shorthand notation can be used to put anything written to file descriptor 2 (**stderr**) in 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.

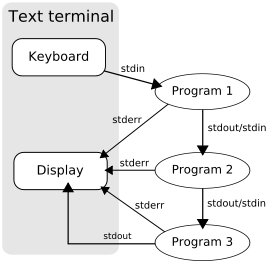
In order to do this we use the vertical-bar, |, (pipe symbol) between commands as in:

$ 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. In addition 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 (Unix)**

From Wikipedia, the free encyclopedia



A pipeline of three programs run on a text terminal

In [Unix-like](http://en.wikipedia.org/wiki/Unix-like) computer [operating systems](http://en.wikipedia.org/wiki/Operating_system) (and, to some extent, [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows)), a **pipeline** is the original [*software pipeline*](http://en.wikipedia.org/wiki/Pipeline_%28software%29): a set of [processes](http://en.wikipedia.org/wiki/Process_%28computing%29) chained by their [standard streams](http://en.wikipedia.org/wiki/Standard_streams), so that the output of each process ([*stdout*](http://en.wikipedia.org/wiki/Stdout)) feeds directly as input ([*stdin*](http://en.wikipedia.org/wiki/Stdin)) to the next one. Each connection is implemented by an [anonymous pipe](http://en.wikipedia.org/wiki/Anonymous_pipe). [Filter programs](http://en.wikipedia.org/wiki/Filter_%28Unix%29) are often used in this configuration.

The concept was invented by [Douglas McIlroy](http://en.wikipedia.org/wiki/Douglas_McIlroy) for [Unix shells](http://en.wikipedia.org/wiki/Unix_shell) and it was named by analogy to a physical [pipeline](http://en.wikipedia.org/wiki/Pipeline_transport).[[1]](http://en.wikipedia.org/wiki/Pipeline_%28Unix%29#cite_note-1) Abstract (illustrated) and concrete examples with the shell syntax:

% program1 | program2 | program3   
 % ls -l | grep key | less

Unix pipeline can be thought of as [left associative](http://en.wikipedia.org/wiki/Left_associative) [infix operation](http://en.wikipedia.org/wiki/Infix_notation) whose operands are programs with parameters. Programatically all programs in pipeline run at the same time (in parallel), but, looking at syntax, it can be thought that one runs after another. It is a [functional composition](http://en.wikipedia.org/wiki/Functional_composition). One can be reminded of [functional programming](http://en.wikipedia.org/wiki/Functional_programming), where data is passed from one function to another (as their input or output).

**Searching for Files**



Being able to quickly find the files you are looking for will make you a much happier Linux user! You can search for files in your parent directory or any other directory on the system as needed.

In this section, you will learn how to use the **locate** and **find** utilities, and how to use **wildcards** in **bash**.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/5aebc93bf9f54c26a6486e0adcdc7e0c/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/5aebc93bf9f54c26a6486e0adcdc7e0c/1#)

**locate**



The **locate** utility program performs a search through 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.

To get a shorter 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:

$ locate zip | grep bin

which will list all 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.

**locate** utilizes the database created by another program, **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.

**Wildcards and Matching File Names**

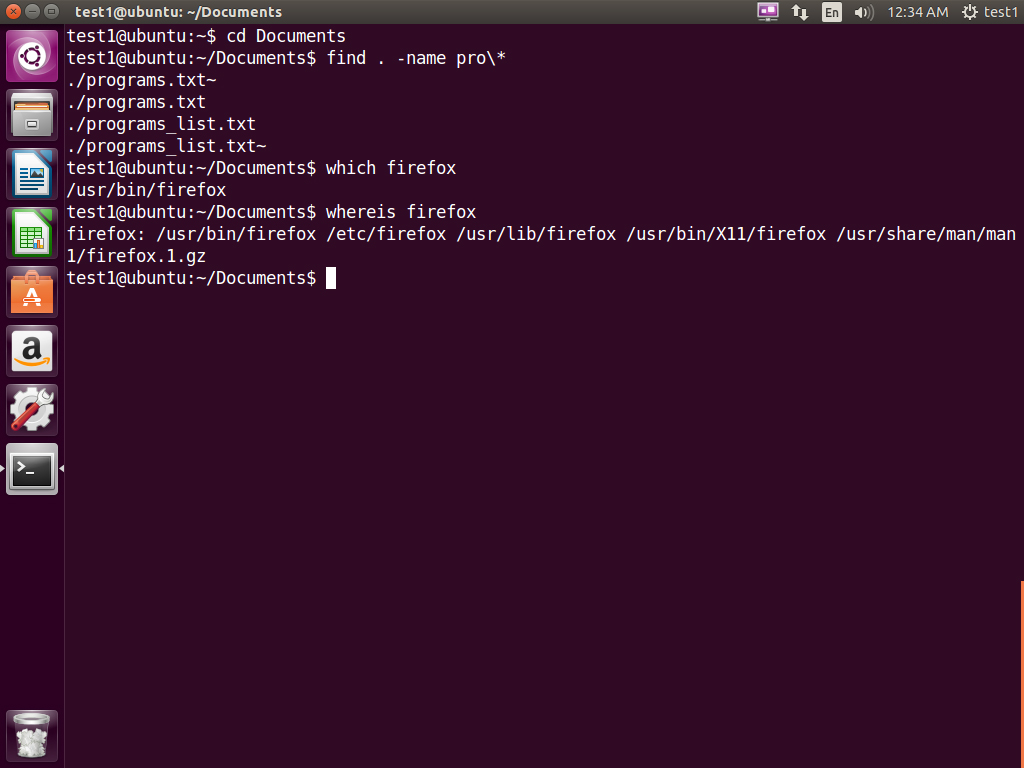
You can search for a filename containing specific characters using **wildcards**.

|  |  |
| --- | --- |
| **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 ?, e.g. if you know only the first 2 letters are 'ba' of a 3-letter filename with an extension of .out, type ls ba?.out.

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

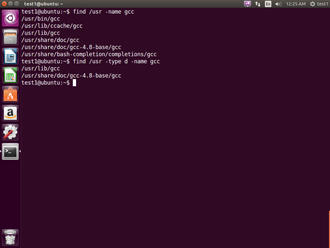
**Finding Files In a Directory**



**find** is 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.

For example, administrators sometimes scan for large **core files** (which contain diagnostic information after a program fails) that are more than several weeks old in order to remove them. It is also common to remove files in **/tmp** (and other temporary directories, such as those containing cached files) that have not been accessed recently. Many distros use automated scripts that run periodically to accomplish such house cleaning.

Using **find**

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).

Searching for files and directories named "gcc":

$ find /usr -name gcc

Searching only for directories named "gcc":

$ find /usr -type d -name gcc

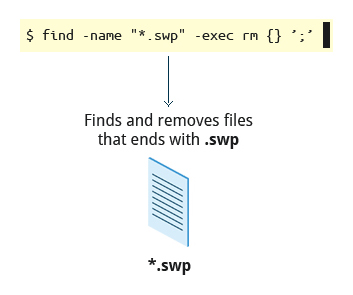
Searching only for regular files named "test1":

$ find /usr -type f -name test1

Click the image to view an enlarged version.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/5aebc93bf9f54c26a6486e0adcdc7e0c/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/5aebc93bf9f54c26a6486e0adcdc7e0c/1#)

**Using Advanced 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:

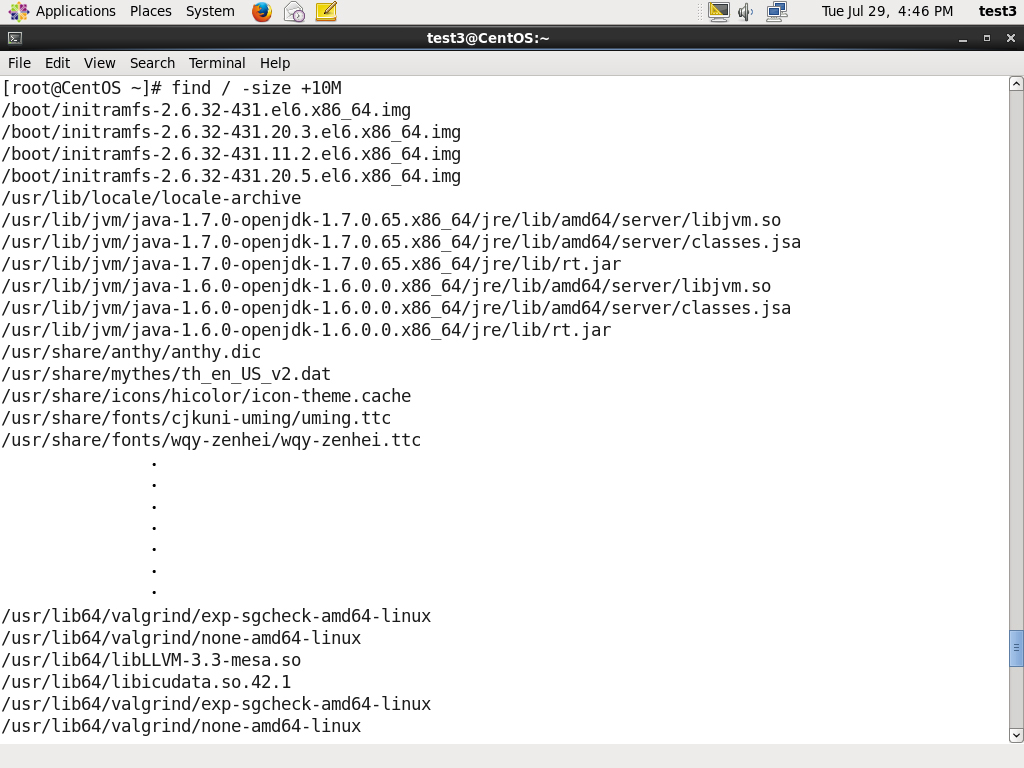
$ find -name "\*.swp" -exec rm {} ’;’

The {} (squiggly brackets) is a place holder 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.

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. Both are easy to accomplish.

Finding based on time:

$ find / -ctime 3

Here, -ctime is when the inode meta-data (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).

Finding 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 {} ’;’

**Working with Files**



Linux provides many commands that help you in viewing the contents of a file, creating a new file or an empty file, changing the **timestamp** of a file, and removing and renaming a file or directory. These commands help you in managing your data and files and in ensuring that the correct data is available at the correct location.

**Viewing Files**

You can use the following utilities to view 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, one line at a time. |
| **less** | Used to view larger files because it is a paging program; it pauses at each screenful 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. |
| **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 and mkdir**



**touch** is often used to set or update the access, change, and modify times of files. By default it resets a file's time stamp to match the current time.

However, you can also create an **empty** file using touch:

$ touch <filename>

This is normally done to create an empty file as a placeholder for a later purpose.

**touch** provides several options, but here is one of interest:

* The -t option allows you to set the date and time stamp of the file.

To set the time stamp to a specific time:

$ touch -t 03201600 myfile

This sets the file, myfile's, time stamp to 4 p.m., March 20th (03 20 1600).

**mkdir** is used to create a directory.

* To create a sample directory named sampdir under the current directory, type mkdir sampdir.
* To create a sample directory called sampdir under /usr, type mkdir /usr/sampdir.

Removing a directory is simply done with **rmdir.** The directory must be empty or it will fail. To remove a directory and all of its contents you have to do rm -rf as we shall discuss.

**Removing a File**

|  |  |
| --- | --- |
| **Command** | **Usage** |
| mv | Rename a file |
| rm | Remove a file |
| rm –f | Forcefully remove a file |
| rm –i | Interactively remove a file |

If you are not certain about removing files that match a pattern you supply, it is always good to run **rm** interactively (rm –i) to prompt before every removal.

**Renaming or Removing a Directory**

**rmdir** works only on empty directories; otherwise you get an error.

While typing rm –rf is a fast and easy way to remove a whole filesystem tree recursively, it is extremely dangerous and should be used with the utmost care, especially when used by root (recall that recursive means drilling down through all sub-directories, all the way down a tree). Below are the commands used to rename or remove a directory:

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

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/ab27cf5d88ae446d9ea2cf73901487f5/#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/ab27cf5d88ae446d9ea2cf73901487f5/#)

**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@quad32 $

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 \$ "

coop@quad64 $ echo $PS1

\u@\h \$

coop@quad64 $

**Note: The next two screen covers a demonstration and Try-It-Yourself activity. You can view a demonstration and practice the Try-It-Yourself activity.**

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/ab27cf5d88ae446d9ea2cf73901487f5/#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/ab27cf5d88ae446d9ea2cf73901487f5/#)

**Package Management Systems on Linux**



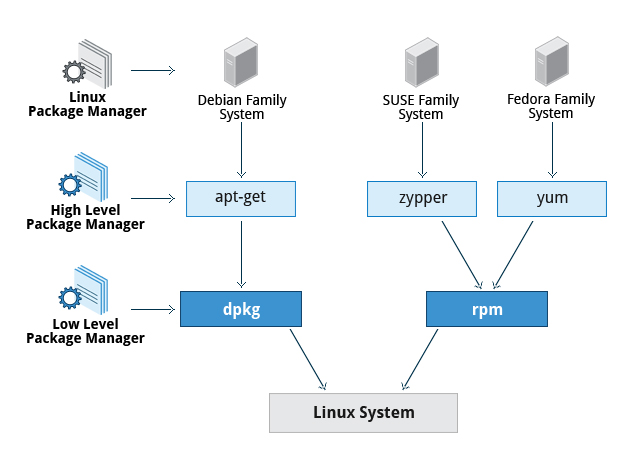
The core parts of a Linux distribution and most of its add-on software are installed via the **Package Management System**. Each package contains the files and other instructions needed to make one software component work on the system. Packages can depend on each other. For example, a package for a 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 provide the same features at a broad level.

In this section, you will learn how to install, remove, or search for packages using the different package management tools.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)

**Package Managers: Two Levels**

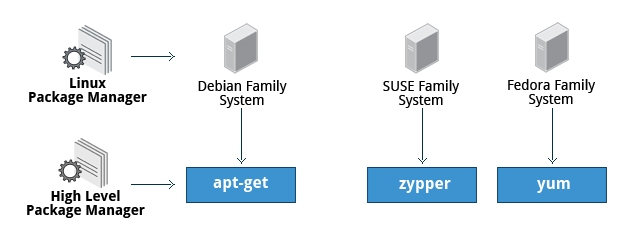


Both package management systems provide two tool levels: a low-level tool (such as **dpkg** or **rpm**), takes care of the details of unpacking individual packages, running scripts, getting the software installed correctly, while a high-level tool (such as **apt-get**, **yum**, or **zypper**) works with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

Most of the time users need work only with the high-level tool, which will take care of calling the low-level tool as needed. Dependency tracking 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.

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)

**Working With Different Package Management Systems**



* The **Advanced Packaging Tool** (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-get and apt-cache.
* **Yellowdog Updater Modified** (**yum**) is an open-source command-line package-management utility for RPM-compatible Linux systems, basically what we have called the **Fedora** family. **yum** has both command line and graphical user interfaces.
* **zypper** is a package management system for **openSUSE** that is 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, click the link below:

[Basic Packaging Commands](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/Basic_Packagaing_Commands.pdf)

**Note: The next few screens cover the demonstrations of a member of each of the three Linux distribution families we cover in this course. You can view a demonstration for the distribution type of your choice.**

* [Previous](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)
* [Next](https://courses.edx.org/courses/LinuxFoundationX/LFS101x/2T2014/courseware/fef2ac9b822744958447641cbe43212c/41827d16d2074d199c13c1e72d9789df/1#)