Module 2 - The command line

By Juan Medina

jmedina@collin.edu

COLLIN COLLEGE

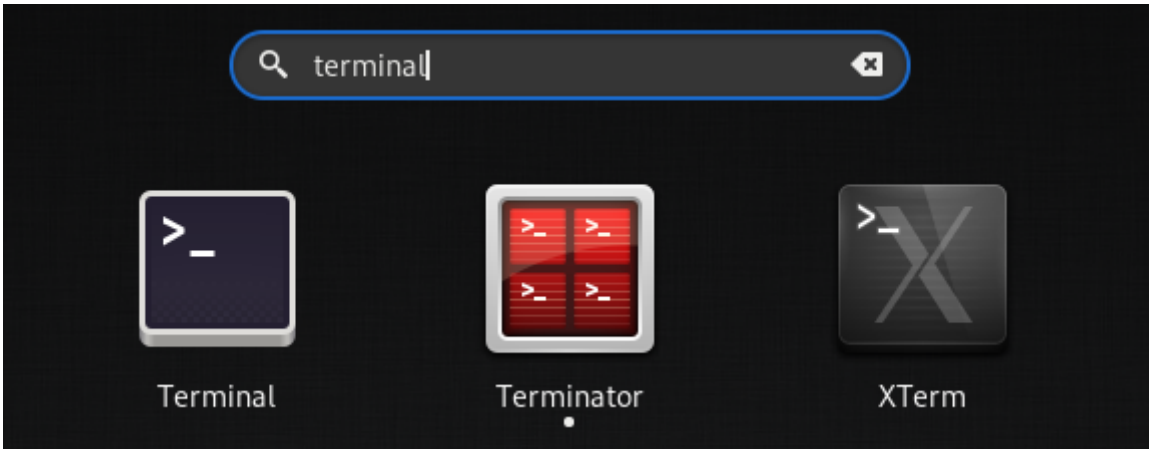
The Command Line

When we speak of the command line, we are really referring to the shell.

The shell is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called bash. The name "bash" is an acronym for "Bourne Again Shell", a reference to the fact bash is an enhanced replacement for sh, the original Unix shell program written by Steve Bourne.

Terminal Emulators

When using a graphical user interface, we need another program called a terminal emulator to interact with the shell. If we look through our desktop menus, we will the gnome-terminal simply "terminal" on our menu.



There are a number of other terminal emulators available for Linux. You will probably develop a preference for one or another based on the number of bells and whistles it has.

Starting a terminal

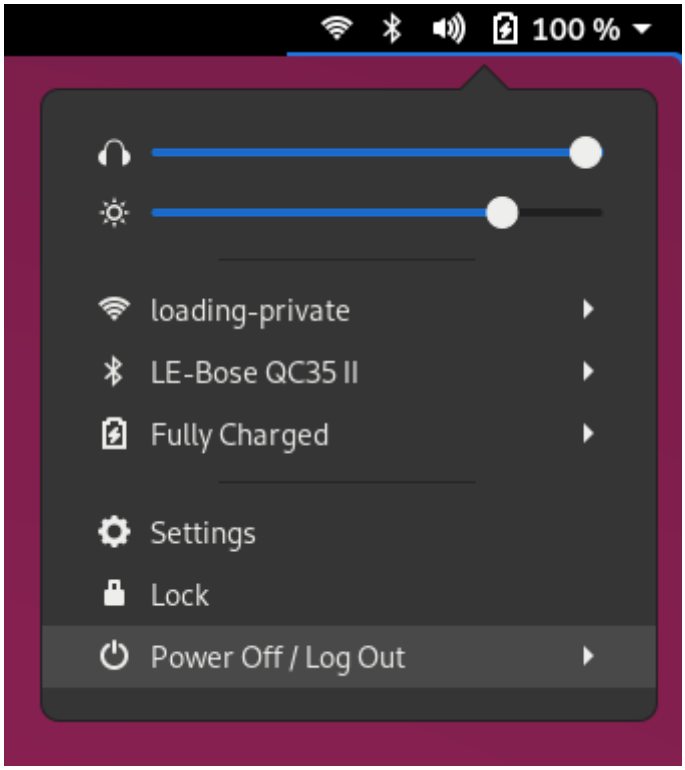
- From the Activities overview, search for the word `terminal`.

When a terminal window is opened, a shell prompt displays for the user that started the graphical terminal program. The shell prompt and the terminal window's title bar indicate:

```
1 | [ `current user name`@`hostname` `current working directory` ]$
```

Locking the screen, logout and shutdown

Locking the screen, logging out entirely or shutdown the computer, can be done from the system menu on the far right of the top bar.



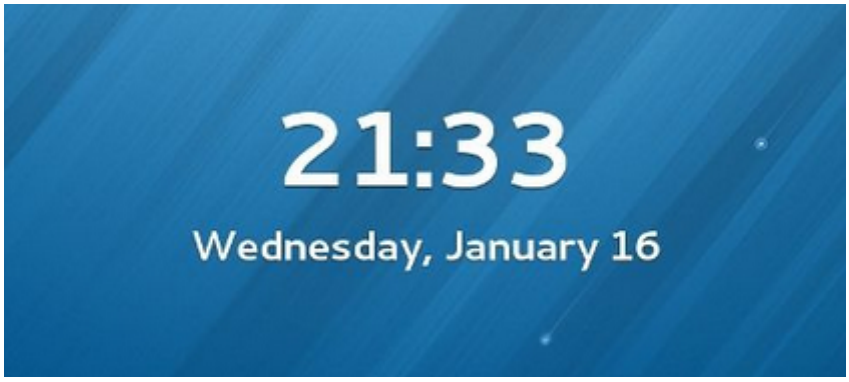
All this actions can also be performed as follows

Lock screen

Press `Super+L`.

The screen also locks if the graphical session is idle for a few minutes (behavior that you can adjust in preferences).

A lock screen curtain appears that shows the system time and the name of the logged-in user.



To unlock the screen, press Enter or Space to raise the lock screen curtain, then enter that user's password on the lock screen.

logout

Select the system menu in the upper-right corner on the top bar and select (User) → Log Out. A window displays that offers the option to Cancel or confirm the Log Out action.

Shutdown

From the system menu in the upper-right corner, click the power button at the bottom of the menu or press Ctrl+Alt+Del. In the dialog box that displays, you can choose to Power Off or Restart the machine, or Cancel the operation. If you do not make a choice, the system automatically shuts down after 60 seconds.

Your First Keystrokes

So let's get started. Launch the terminal emulator! Once it comes up, we should see something like this:

```
1 [jmedinar@localhost ~]$
```

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 usually include your `username@machinename`, followed by the current working directory and a dollar sign. If the last character of the prompt is a pound sign ("#") rather than a dollar sign, the terminal session has superuser privileges. This means either we are logged in as the root user or we selected a terminal emulator that provides superuser privileges.

Assuming that things are good so far, let's try some typing. Enter some gibberish at the prompt like so:

```
1 [jmedinar@localhost ~]$ gfgdgfdgd
2 bash: gfgdgfdgd: command not found...
3 [jmedinar@localhost ~]$
```

Since this command makes no sense, the shell will tell us so and give us another chance

Shell basics

Commands entered at the shell prompt have three basic parts:

- Command to run
- Options to adjust the behavior of the command
- Arguments, which are typically targets of the command

The command is the name of the program to run. It may be followed by one or more options, which adjust the behavior of the command or what it will do. Options normally start with one or two dashes (`-a` or `--all`, for example) to distinguish them from arguments. Commands may also be followed by one or more arguments, which often indicate a target that the command should operate upon.

When you are ready to execute a command, press the Enter key. Type each command on a separate line. The command output is displayed before the next shell prompt appears.

```
[jmedinar@localhost ~]$ whoami
jmedinar
[jmedinar@localhost ~]$
```

If you want to type more than one command on a single line, use the semicolon (;) as a command separator. In this case the output of both commands will be displayed before the next shell prompt appears.

```
[jmedinar@localhost ~]$ whoami; date
jmedinar
Sat Dec 19 08:53:35 PM CST 2020
[jmedinar@localhost ~]$
```

The date command displays the current date and time. It can also be used by the superuser to set the system clock. An argument that begins with a plus sign (+) specifies a format string for the date command.

```
[jmedinar@localhost ~]$ date +%R
20:54
[jmedinar@localhost ~]$ date +%x
12/19/2020
```

Linux does not require file name extensions to classify files by type. The file command scans the beginning of a file's contents and displays what type it is. The files to be classified are passed as arguments to the command.

```
[jmedinar@localhost ~]$ file /etc/passwd
/etc/passwd: ASCII text
[jmedinar@localhost ~]$ file /bin/passwd
/bin/passwd: setuid ELF 64-bit LSB pie executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=7855a717a348551b97c8bc1c45f4ee92d238c9f1, for
GNU/Linux 3.2.0, stripped
[jmedinar@localhost ~]$ file /home
/home: directory
```

Logging in from a terminal

Every time you run a shell you are effectively login in to the computer on a terminal.

Besides the `Terminal Emulators` Linux also provides multiple `Virtual Consoles`, which can run separate terminals on their own. Each virtual console supports an independent login session and you can switch between them by pressing Ctrl+Alt and a function key from `F1` through `F6` at the same time.

Logging out from a terminal

When you are finished using the shell and want to quit, you can choose one of several ways to end the session. You can enter the exit command to terminate the current shell session `exit`. Alternatively, finish a session by pressing `Ctrl+D`.

The following is an example of a user logging out of an SSH session:

```
[jmedinar@localhost ~]$ exit
logout
Connection to remotehost closed.
[jmedinar@localhost ~]$
```

Viewing the content of files

One of the most simple and frequently used commands in Linux is `cat`. The `cat` command allows you to read single or multiple files. The example shows how to view the contents of the `/etc/passwd` file.

```
[jmedinar@localhost ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
...output omitted...
```

Use the following command to display the contents of multiple files.

```
[jmedinar@localhost ~]$ cat file1 file2
Hello World!!
Introduction to Linux commands.
```

Some files are very long and can take up more room to display than that provided by the terminal. The cat command does not display the contents of a file as pages. The `less` command displays one page of a file at a time and lets you scroll at your leisure. The `less` command allows you to page forward and backward through files that are longer than can fit on one terminal window. Use the UpArrow key and the DownArrow key to scroll up and down. Press Q to exit the command.

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

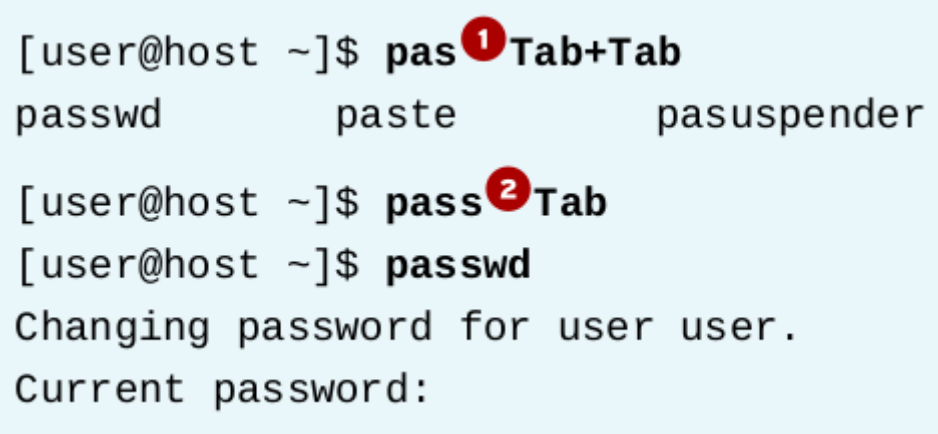
```
[jmedinar@localhost ~]$ head -n 3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[jmedinar@localhost ~]$ tail -n 3 /etc/passwd
vboxadd:x:978:1::/var/run/vboxadd:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
jmedinar:x:1000:1000:Juan Medina:/home/jmedinar:/bin/bash
```

The `wc` command counts lines, words, and characters in a file. It takes a `-l`, `-w`, or `-c` option to display only the number of lines, words, or characters, respectively.

```
[jmedinar@localhost ~]$ wc -l /etc/passwd /etc/group /etc/hosts
  48 /etc/passwd
  84 /etc/group
   5 /etc/hosts
 137 total
[jmedinar@localhost ~]$ wc -c /etc/passwd /etc/group /etc/hosts
2662 /etc/passwd
1201 /etc/group
 183 /etc/hosts
4046 total
[jmedinar@localhost ~]$ wc -w /etc/passwd /etc/group /etc/hosts
 112 /etc/passwd
  84 /etc/group
  12 /etc/hosts
 208 total
```

Tab completion

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




```
Press Tab twice: 1
Press Tab once: 2
```

Tab completion can be used to complete file names when typing them as arguments to commands. When Tab is pressed, it completes as much of the file name as possible. Pressing Tab a second time causes the shell to list all of the files that are matched by the current pattern. Type additional characters until the name is unique, then use tab completion to complete the command.

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

```
[root@localhost ~]# useradd --[TAB+TAB]
--badnames               --gid                    --no-log-init
    --shell
--base-dir               --groups                 --non-unique
   --skel
--btrfs-subvolume-home   --help                   --no-user-group
    --system
--comment                --home-dir               --password
   --uid
--create-home            --inactive               --prefix
   --user-group
--defaults               --key                    --root

--expiredate             --no-create-home         --selinux-user
```

Multi-line commands

Commands that support many options and arguments can quickly grow quite long and are automatically scrolled by the Bash shell. As soon as the cursor reaches the right margin of the window, the command continues on the next line. To make the readability of the command easier, you can break it up so that if fits on more than one line.

To do this, add a backslash character `\` as the last character on the line. This tells the shell to ignore the newline character and treat the next line as if it were part of the current line.

```
[jmedinar@localhost ~]$ head -n 3 \
> /etc/passwd \
> /etc/group
==> /etc/passwd <==
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin

==> /etc/group <==
root:x:0:
bin:x:1:
daemon:x:2:
```

Command History

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

```
[user@host ~]$ history
   ...output omitted...
   23 clear
   24 who
   25 pwd
   26 ls /etc
   27 uptime
   28 ls
   29 date
   30 history
```

The exclamation point character (!) is a meta-character that is used to expand previous commands without having to retype them. The `!number` command expands to the command matching the number specified. The !string command expands to the most recent command that begins with the string specified.

```
[jmedinar@localhost ~]$ !28
ls
 bin       Documents   GIT     Pictures   Templates   Videos
   vm_remote
```

The arrow keys can be used to navigate through previous commands in the shell's history.

`UpArrow` edits the previous command in the history list. `DownArrow` edits the next command in the history list. `LeftArrow` and `RightArrow` move the cursor left and right in the current command from the history list, so that you can edit it before running it.

Editing the command line

Bash has some command-line editing features. This allows the user to use text editor commands to move around within and modify the current command being typed. Besides using the arrow keys. More powerful editing commands are introduced in the following table.

SHORTCUT	DESCRIPTION
Ctrl+A	Jump to the beginning of the command line.
Ctrl+E	Jump to the end of the command line.
Ctrl+U	Clear from the cursor to the beginning of the command line.
Ctrl+K	Clear from the cursor to the end of the command line.
Ctrl+LeftArrow	Jump to the beginning of the previous word on the command line.

SHORTCUT	DESCRIPTION
Ctrl+RightArrow	Jump to the end of the next word on the command line.
Ctrl+R	Search the history list of commands for a pattern.

Learn your first Commands

Now that we have learned to type, let's try a few simple commands that will give us some information about our system.

Command	Description
`date`	print or set the system date and time
`cal`	display a calendar
`df`	report file system disk space usage
`free`	Display amount of free and used memory in the system
`uptime`	Tell how long the system has been running.
`ps`	report a snapshot of the current processes.
`echo`	display a line of text
`ls`	list directory contents
`clear`	clear the terminal screen
`pwd`	print name of current/working directory
`touch`	change file timestamps and create files
`w`	Show who is logged on and what they are doing.
`who`	show who is logged on
`whoami`	print effective userid
`last`	show a listing of last logged in users
`hostname`	show or set the system's hostname
`exit`	Exit the shell.

Try them all!

What Exactly Are Commands?

A command can be one of four different things:

1. An executable program like all those files we saw in /usr/bin.
2. A command built into the shell itself.
3. A shell function.
4. An alias. Commands that we can define ourselves, built from other commands.

Asking for Help!

We have seen a series of mysterious commands, each with its own mysterious options and arguments. Let's remove some of that mystery with some commands that will allow us to understand better what is going on.

type

The type command is a shell built-in that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
[jmedinar@localhost ~]$ type ls
ls is aliased to `ls --color=auto'
[jmedinar@localhost ~]$ type date
date is hashed (/usr/bin/date)
[jmedinar@localhost ~]$ type passwd
passwd is /usr/bin/passwd
```

which

Display which executable program will be executed. Sometimes there is more than one version of an executable program installed on a system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the which command is used:

```
[jmedinar@localhost ~]$ which ls
alias ls='ls --color=auto'
   /usr/bin/ls
[jmedinar@localhost ~]$ which date
/usr/bin/date
[jmedinar@localhost ~]$ which passwd
/usr/bin/passwd
```

whatis

Display a very brief description of a command.

```
[jmedinar@localhost ~]$ whatis ls
ls (1)               - list directory contents
ls (1p)              - list directory contents
[jmedinar@localhost ~]$ whatis date
date (1)             - print or set the system date and time
date (1p)            - write the date and time
[jmedinar@localhost ~]$ whatis passwd
openssl-passwd (1ssl) - compute password hashes
passwd (1)           - update user's authentication tokens
passwd (5)           - password file
```

apropos

Display a list of appropriate commands. It is also possible to search the list of man pages for possible matches based on a search term. It's very crude but sometimes helpful.

```
[jmedinar@localhost ~]$ apropos password
chage (1)            - change user password expiry information
chgpasswd (8)        - update group passwords in batch mode
chpasswd (8)         - update passwords in batch mode
cracklib-check (8)   - Check passwords using libcrack2
```

The first field in each line of output is the name of the man page, the second field shows the section.

help

bash has a built-in help facility available for each of the shell built-ins. To use it, type `help` followed by the name of the shell built-in. For example:

```
[jmedinar@localhost ~]$ help cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the
value of the
    HOME shell variable.

    The variable CDPATH defines the search path for the directory
containing
    DIR.  Alternative directory names in CDPATH are separated by a
colon (:).

... and more information ...
```

A note on notation: When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
cd: cd [-L|[-P [-e]] [-@]] [dir]
```

This notation says that the command `cd` may be followed optionally by either a `-L` or a `-P` and further, if the `-P` option is specified the "`-e`" option may also be included followed by the optional argument `dir`.

- -help

Get help for shell built-in commands

```
[jmedinar@localhost ~]$ passwd --help
Usage: passwd [OPTION...] <accountName>
  -k, --keep-tokens       keep non-expired authentication tokens
  -d, --delete            delete the password for the named account
(root only); also removes password
                          lock if any
  -l, --lock              lock the password for the named account
(root only)
  -u, --unlock            unlock the password for the named account
(root only)
  -e, --expire            expire the password for the named account
(root only)
  -f, --force             force operation
  -x, --maximum=DAYS      maximum password lifetime (root only)
  -n, --minimum=DAYS      minimum password lifetime (root only)
  -w, --warning=DAYS      number of days warning users receives
before password expiration (root only)
  -i, --inactive=DAYS     number of days after password expiration
when an account becomes disabled
                          (root only)
  -S, --status            report password status on the named
account (root only)
      --stdin             read new tokens from stdin (root only)
```

Some programs don't support the `--help` option but try it anyway. Often it results in an error message that will reveal the same or similar usage information.

man pages

One source of documentation that is generally available on the local system are system manual pages or man pages. These pages are shipped as part of the software packages for which they provide documentation, and can be accessed from the command line by using the man command.

```
[jmedinar@localhost ~]$ man ls
```

```
LS(1)                                          User Commands
                                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

```

```
DESCRIPTION
       List  information  about  the FILEs (the current directory by default).  Sort entries alphabeti-
       cally if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.
       -a, --all
              do not ignore entries starting with .
       -A, --almost-all
              do not list implied . and ..
       --author
              with -l, print the author of each file
... Output truncated ...
```

Navigate and search man pages

The ability to efficiently search for topics and navigate man pages is a critical administration skill. GUI tools make it easy to configure common system resources, but using the command-line interface is still more efficient. To effectively navigate the command line, you must be able to find the information you need in man pages. The following table lists basic navigation commands when viewing man pages:

COMMAND	RESULT
`PageDown`	Scroll forward (down) one screen
`PageUp`	Scroll backward (up) one screen
`DownArrow`	Scroll forward (down) one line
`UpArrow`	Scroll backward (up) one line
`` `/string ``	Search forward (down) for string in the man page
`N`	Repeat previous search forward (down) in the man page
`Shift+N`	Repeat previous search backward (up) in the man page
`G`	Go to start of the man page.
`Shift+G`	Go to end of the man page.
`Q`	Exit man and return to the command shell prompt

Reading Man Pages

Each topic is separated into several parts. Most topics share the same headings and are presented in the same order. Typically a topic does not feature all headings, because not all headings apply for all topics.

Common headings are:

HEADING	DESCRIPTION
NAME	Subject name. Usually a command or file name. Very brief description.
SYNOPSIS	Summary of the command syntax.
DESCRIPTION	In-depth description to provide a basic understanding of the topic.
OPTIONS	Explanation of the command execution options.
EXAMPLES	Examples of how to use the command, function, or file.
FILES	A list of files and directories related to the man page.
SEE ALSO	Related information, normally other man page topics.
BUGS	Known bugs in the software.
AUTHOR	Information about who has contributed to the development of the topic.

Searching for man pages by keyword

A keyword search of man pages is performed with man -k keyword, which displays a list of keyword-matching man page topics with section numbers.

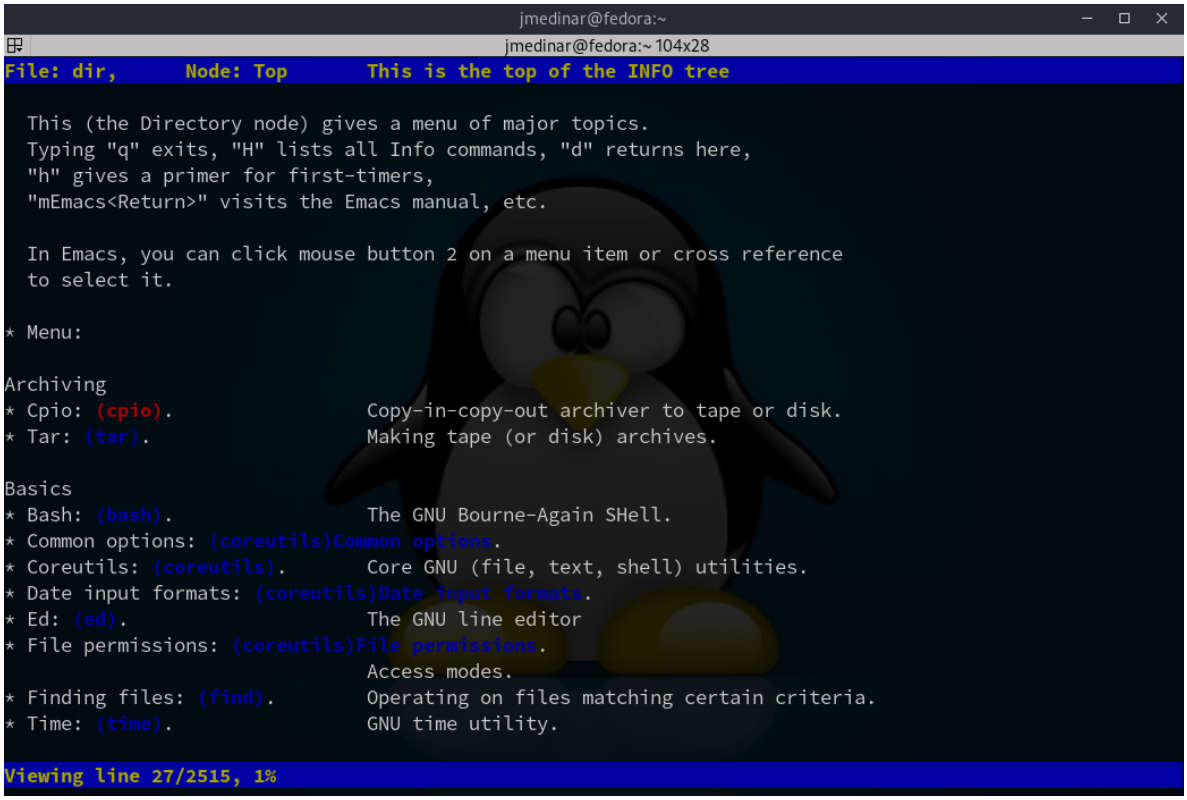
```
[jmedinar@localhost ~]$ man -k passwd
chgpasswd (8)        - update group passwords in batch mode
chpasswd (8)         - update passwords in batch mode
passwd (1)           - update user authentication tokens
passwd (5)           - password file
```

Popular system administration topics are in sections 1 (user commands), 5 (file formats), and 8 (administrative commands). Administrators using certain troubleshooting tools also use section 2 (system calls). The remaining sections are generally for programmer reference or advanced administration.

Info documentation

Man pages have a format useful as a command reference, but less useful as general documentation. For such documents, the GNU Project developed a different on-line documentation system, known as GNU Info. Info documents are an important resource on a Linux system because many fundamental components and utilities are either developed by the GNU Project or utilize the Info document system.

To launch the Info document viewer, use the `pinfo` command.



Info documentation is comprehensive and hyper-linked. It is possible to output info pages to multiple formats. By contrast, man pages are optimized for printed output. The Info format is more flexible than man pages, allowing thorough discussion of complex commands and concepts. Like man pages, Info nodes are read from the command line, using the `pinfo` command.

A typical man page has a small amount of content focusing on one particular topic, command, tool, or file. The Info documentation is a comprehensive document. Info provides the following improvements:

- One single document for a large system containing all the necessary information for that system
- Hyper-links
- A complete browsable document index
- A full text search of the entire document

Some commands and utilities have both man pages and info documentation; usually, the Info documentation is more in depth.

Compare the differences in tar documentation using man and pinfo:

```
[jmedinar@localhost ~]$ man tar
[jmedinar@localhost ~]$ pinfo tar
```

The `pinfo` reader is more advanced than the original info command. To browse a specific topic, use the `pinfo` topic command.

The `pinfo` command without an argument opens the top directory. New documentation becomes available in `pinfo` when their software packages are installed.

Creating Your Own Commands With alias

Now for our very first experience with programming! We will create a command of our own using the alias command.

The first thing we have to do is dream up a name for our new command. Let's try "**test**". Before we do that, it would be a good idea to find out if the name "**test**" is already being used. To find out, we can use the **type** command again:

```
1 [jmedinar@localhost ~]$ type test
2 test is a shell builtin
```

Oops! The name "test" is already taken. Let's try `mytest`:

```
1 [jmedinar@localhost ~]$ type mytest
2 bash: type: mytest: not found
```

Great! `mytest` is not taken. So let's create our alias: Notice the structure of this command:

```
1 [jmedinar@localhost ~]$ alias mytest='pwd; cd /usr; pwd; ls; cd -;'
```

After the command `alias` we give alias a name followed immediately (no whitespace allowed) by an equals sign `=`, followed immediately by a quoted string containing the meaning to be assigned to the name.

After we define our alias, it can be used anywhere the shell would expect a command. Let's try it our new command:

```
1 [jmedinar@localhost ~]$ mytest
2 /home/jmedinar
3 /usr
4 bin  games  include  lib  lib64  libexec  local  sbin  share  src
  tmp
5 /home/jmedinar
```

To remove an alias, the `unalias` command is used, like so:

```
1 [jmedinar@localhost ~]$ unalias mytest
```

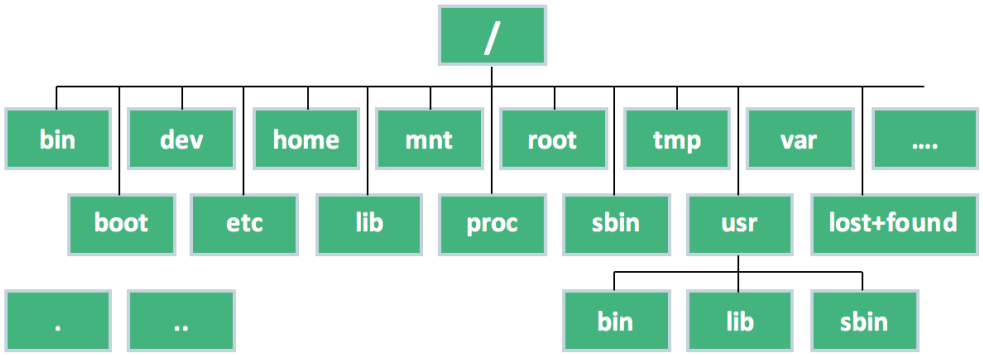
Navigating Directories & Files

The first thing we need to learn is how to navigate the file system on our Linux system.

Understanding The File System Tree

Like Windows, Linux organizes its files in what is called a hierarchical directory structure. This means that they are organized in a tree-like pattern of directories, 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 and so on. Note that unlike Windows, which has a separate file system tree for each storage device, Linux always has a single file system tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached at various points on the tree according to the whims of the system administrator, the person responsible for the maintenance of the system.

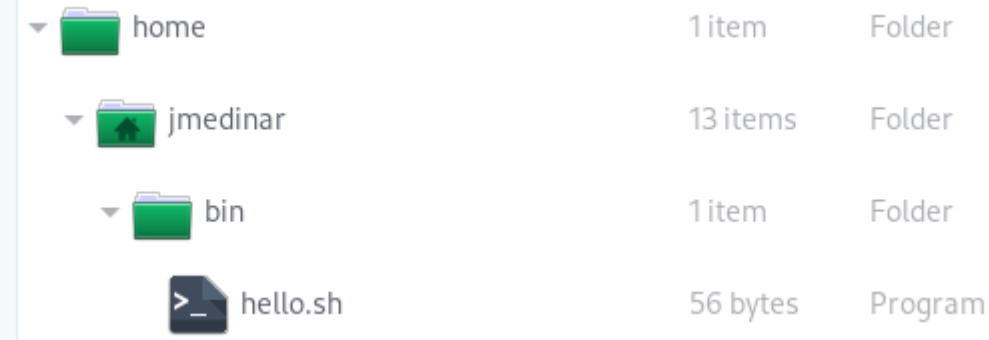


- **/:** The root directory. Where everything begins.
- **/bin** : All the executable binary programs (file) required during booting, repairing, files required to run into single-user-mode, and other important, basic commands
- **/boot** : Holds important files during the boot-up process, including **Linux Kernel**.
- **/dev** : Contains device files for all the hardware devices on the machine.
- **/etc** : Contains Application's configuration files, **startup**, **shutdown**, **start**, **stop** script for every individual program.
- **/home** : Home directory of the users.
- **/lib** : The Lib directory contains **kernel modules** and **shared library** images.
- **/lost+found** : This Directory is installed during installation of **Linux**, useful for recovering files that may be broken due to unexpected **shut-down**.

- **/media** : Temporary mount directory is created for removable devices.
- **/mnt** : Temporary mount directory
- **/opt** : Optional is abbreviated as opt. Contains third party application software.
- **/proc** : A virtual and pseudo-file-system which contains information about the running process with a particular Process-id
- **/root** : This is the home directory of root user and **should never be confused with '/'**
- **/sbin** : Contains binary executable programs, required by **System Administrator**
- **/srv** : This directory contains server-specific and service-related files.
- **/sys** : Modern Linux distributions include a **/sys** directory as a **virtual filesystem**, which stores and allows modification of the devices connected to the system.
- **/tmp** :System's Temporary Directory, Accessible by users and root. Stores temporary files for **user** and **system**, till next boot.
- **/usr** : Contains executable **binaries**, **documentation**, **source code**, **libraries** of programs not part of the Operative System.
- **/var** : This directory contains **logs**, **lock**, **spool**, **mail** and **temp** files.

The Current Working Directory

Most of us are probably familiar with a graphical file manager which represents the file system tree as icons.



Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below. However, the command line has no pictures, so to navigate the file system tree we need to think of it in a different way.

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 and any subdirectories below us. 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.

```
1 [jmedinar@localhost ~]$ pwd
2 /home/jmedinar
```

When a user logs in and opens a command window, the initial location is normally the user's home directory. System processes also have an initial directory. Users and processes navigate to other directories as needed; the terms working directory or current working directory refer to their current location.

Listing The Contents Of A Directory

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

```
[jmedinar@localhost ~]$ ls
Desktop Documents Downloads Music Pictures Public Templates Videos
```

We can also get a long listing format with a bit more information

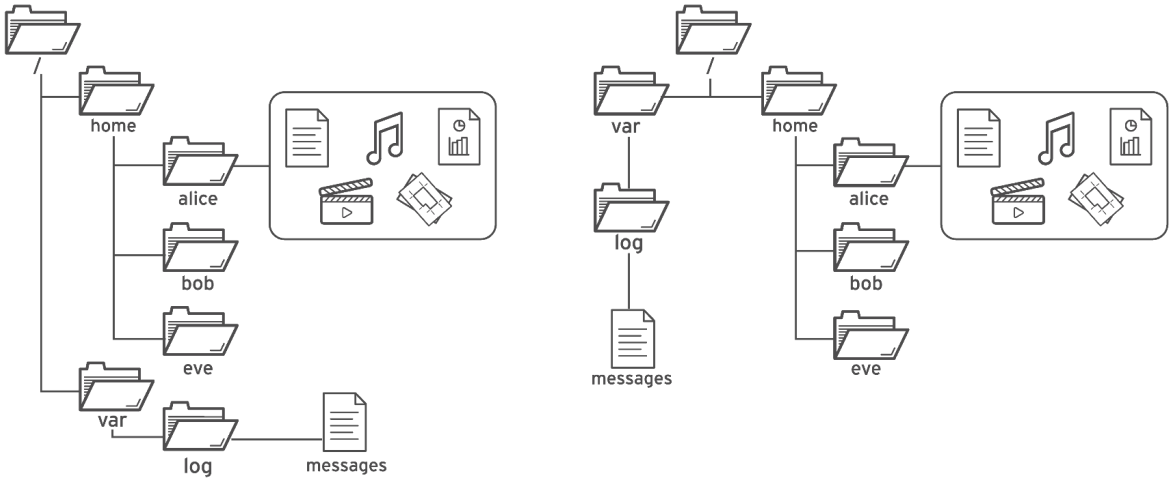
```
[jmedinar@localhost ~]$ ls -l
total 52
drwxr-xr-x. 2 jmedinar jmedinar 4096 Feb 27  2020  Desktop
drwxr-xr-x. 7 jmedinar jmedinar 4096 Oct  4 21:03  Documents
drwxr-xr-x. 5 jmedinar jmedinar 4096 Dec 19 15:41  Downloads
drwxr-xr-x. 3 jmedinar jmedinar 4096 Aug  3 18:02  Music
drwxr-xr-x. 5 jmedinar jmedinar 4096 Dec  2 18:54  Pictures
drwxr-xr-x. 3 jmedinar jmedinar 4096 Apr  7  2020  Public
drwxr-xr-x. 2 jmedinar jmedinar 4096 Feb 27  2020  Templates
drwxr-xr-x. 3 jmedinar jmedinar 4096 Aug  3 18:02  Videos
```

Looking at the Directory tree

To see the directory tree in a similar way we do it in the graphical interface we can use the command tree.

```
[jmedinar@localhost ~]$ tree
.
├── Desktop
├── Documents
│   ├── Code
│   │   ├── file1
│   │   ├── file2
```

ABSOLUTE PATHS AND RELATIVE PATHS



The path of a file or directory specifies its unique file system location. Following a file path traverses one or more named subdirectories, delimited by a forward slash `/`, until the destination is reached. Directories, also called folders, contain other files and other subdirectories. They can be referenced in the same

manner as files.

Absolute Pathnames or Full Path

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. (We know it begins with the root directory when we see the path starting with a `/` symbol)

```
[jmedinar@localhost ~]$ ls -l /etc/sysconfig/network-scripts/ifcfg-CougarWiFi
-rw-r--r--. 1 root root 350 Sep 12 15:47 /etc/sysconfig/network-scripts/ifcfg-CougarWiFi
```

Relative Pathnames

Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the current working directory. To do this, it uses a couple of special symbols to represent relative positions in the file system tree.

```
"." (dot)
# and
".." (dot dot).
```

The "." symbol refers to the working directory and the ".." symbol refers to the working directory's parent directory. Here is how it works. Let's change the working directory to `/usr/bin`:

```
[jmedinar@localhost ~]$ cd /usr/bin
[jmedinar@localhost bin]$ pwd
/usr/bin
```

Okay, 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 with an absolute pathname:

```
[jmedinar@localhost bin]$ cd /usr
[jmedinar@localhost usr]$ pwd
/usr
```

Or, with a relative pathname:

```
[jmedinar@localhost bin]$ cd ..
[jmedinar@localhost usr]$ pwd
/usr
```

Like an absolute path, a relative path identifies a unique file, specifying only the path necessary to reach the file from the working directory. Recognizing relative path names follows a simple rule: A path name with anything other than a forward slash as the first character is a relative path name. A user in the`/var`

directory could refer to the message log file relatively as `log/messages`.

Some Helpful Shortcuts

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.

Determining A File's Type With file

As we explore the system it will be useful to know what files contain. To do this we will use the file command to determine a file's type. Filenames in Linux are not required to reflect a file's contents. While a filename like `collin.png` would normally be expected to contain a PNG image, it is not required to in Linux. We can invoke the file command this way:

```
1 [jmedinar@localhost Pictures]$ file collin.png
2 collin.png: PNG image data, 1000 x 1000, 8-bit/color RGBA, non-
  interlaced
```

There are many kinds of files. In fact, one of the common ideas in Linux is that "everything is a file." We will see just how true that statement is. While many of the files on your system are familiar, for example, MP3 and JPEG, there are many kinds that are a little less obvious and a few that are quite strange.

Find, grep and pipes

The 3 commands we will explore here are some of the most powerful commands you can learn and to get to fully understand them you will have to acquire some practice and be patient. We will explore their basic usage and some of the most common scenarios but we will be just scratching the surface of the possibilities.

Before we get into all the possibilities of the `find` command we have to discuss it's old brother `locate`

locate – Find Files The Easy Way

The locate command finds files based on the name or path to the file. It is fast because it looks up this information from the `mlocate` database. However, this database is not updated in real time, and it must be frequently updated for results to be accurate. This also means that locate will not find files that have been created since the last update of the database. The locate database is automatically updated every day. However, at any time the root user can issue the `updatedb` command to force an immediate update.

```
[root@host ~]# updatedb
```

The locate program performs a rapid database search of pathnames and then outputs every name that matches a given substring. Say, for example, we want to find all the programs with names that begin with "zip." Since we are looking for programs, we can assume that the name of the directory containing the programs would end with "bin/". Therefore, we could try to use locate this way to find our files:

```
[jmedinar@localhost ~]$ locate bin/zip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipdetails
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

locate will search its database of pathnames and output any that contain the string and If the search requirement is not so simple, locate can be combined with other tools such as `grep` to design more interesting searches. We will discuss `grep` in this module.

```
[jmedinar@localhost ~]$ locate bin/zip | grep info
/usr/bin/zipinfo
```

The -i option performs a case-insensitive search. With this option, all possible combinations of upper and lowercase letters match the search.

```
[jmedinar@localhost ~]$ locate -i messages
...output omitted...
/usr/share/vim/vim80/lang/zh_TW/LC_MESSAGES
/usr/share/vim/vim80/lang/zh_TW/LC_MESSAGES/vim.mo
/usr/share/vim/vim80/lang/zh_TW.UTF-8/LC_MESSAGES
/usr/share/vim/vim80/lang/zh_TW.UTF-8/LC_MESSAGES/vim.mo
/usr/share/vim/vim80/syntax/messages.vim
/usr/share/vim/vim80/syntax/msmessages.vim
/var/log/messages
```

The -n option limits the number of returned search results by the locate command. The following example limits the search results returned by locate to the first five matches:

```
[jmedinar@localhost ~]$ locate -n 5 snow.png
/home/jmedinar/.icons/Arc-OSX-Icons-2.1/src/Paper/16x16/status/stock_weather-snow.png
/home/jmedinar/.icons/Arc-OSX-Icons-2.1/src/Paper/16x16/status/weather-snow.png
/home/jmedinar/.icons/Arc-OSX-Icons-2.1/src/Paper/16x16@2x/status/stock_weather-snow.png
/home/jmedinar/.icons/Arc-OSX-Icons-2.1/src/Paper/16x16@2x/status/weather-snow.png
/home/jmedinar/.icons/Arc-OSX-Icons-2.1/src/Paper/24x24/status/stock_weather-snow.png
```

find – Find Files The Hard Way

While the locate program can find a file-based solely on its name, the find program searches a given directory and its subdirectories for files based on a HUGE variety of attributes. We're going to spend a lot of time with the find command because it has a lot of interesting features that we will see again and again.

In its simplest use, find is given one or more names of directories to search. For example, to produce a list of our home directory:

```
[jmedinar@localhost ~]$ find
```

On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let's use the command `wc` to count the number of files:

```
[jmedinar@localhost ~]$ find | wc -l
502710
```

Wow, we've been busy! The beauty of find is that it can be used to identify files that meet specific criteria. It does this through the application of options, tests, and actions

The find command locates files by performing a real-time search in the file-system hierarchy. It is slower than locate, but more accurate. It can also search for files based on criteria other than the file name, such as the permissions of the file, type of file, its size, or its modification time.

The find command looks at files in the file system using the user account that executed the search. The user invoking the find command must have read and execute permission on a directory to examine its contents.

The first argument to the find command is the directory to search. If the directory argument is omitted, find starts the search in the current directory and looks for matches in any subdirectory.

Searching by name or extension

To search for files by file name, use the `-name FILENAME` option. With this option, find returns the path to files matching `FILENAME` exactly. For example, to search for files named `sshd_config` starting from the `/` directory, run the following command:

```
[root@localhost ~]# find / -name sshd_config
/etc/ssh/sshd_config
```

> *NOTE*
>
> *With the find command, the full word options use a single dash and options follow the path name argument, unlike most other Linux commands.*

Wild-cards are available to search for a file name and return all results that are a partial match.

When using wild-cards, it is important to quote the file name to look for to prevent the terminal from interpreting the wild-card.

In the following example, search for files starting in the `/` directory that have a name ending in `.txt`:

```
[root@localhost ~]# find / -name '*.txt'
/opt/pdfsam-4.1.4-linux/doc/third-party/sejda-io/LICENSE.txt
/opt/pdfsam-4.1.4-linux/doc/third-party/bouncy-castle/LICENSE.txt
/opt/pdfsam-4.1.4-linux/doc/third-party/hibernate-
validator/LICENSE.txt
...output omitted...
```

To search for files in the /etc/ directory that contain the word, `pass`, anywhere in their names on host, run the following command:

```
[root@localhost ~]# find /etc -name '*pass*'
/etc/passwd
/etc/pam.d/password-auth
/etc/pam.d/passwd
...output omitted...
```

To perform a case-insensitive search for a given file name, use the `-iname` option, followed by the file name to search. To search files with case-insensitive text, messages, in their names in the / directory on host, run the following command:

```
[root@localhost ~]# find / -iname '*messages*'
/usr/share/vim/vim80/lang/zh_CN.UTF-8/LC_MESSAGES
/usr/share/vim/vim80/lang/zh_CN.cp936/LC_MESSAGES
/usr/share/vim/vim80/lang/zh_TW/LC_MESSAGES
/usr/share/vim/vim80/lang/zh_TW.UTF-8/LC_MESSAGES
/usr/share/vim/vim80/syntax/messages.vim
/usr/share/vim/vim80/syntax/msmessages.vim
...output omitted...
```

Searching by ownership or permission

The find command can search for files based on their ownership or permissions. Useful options when searching by owner are `-user` and `-group`, which search by name, and `-uid` and `-gid`, which search by ID.

Search for files owned by a user called `jmedinar` in the current directory.

```
[jmedinar@localhost ~]$ find -user jmedinar
.
./.gnupg
./.gnupg/pubring.kbx
./.gnupg/trustdb.gpg
./.gnome2
./.themes
...output omitted...
```

Search for files owned by the group `jmedinar` in the current directory.

```
[jmedinar@localhost ~]$ find -group jmedinar
.
./.gnupg
./.gnupg/pubring.kbx
./.gnupg/trustdb.gpg
./.gnome2
./.themes
...output omitted...
```

Search for files owned by user ID 1000 in the current directory.

```
[jmedinar@localhost ~]$ find -uid 1000
.
./.gnupg
./.gnupg/pubring.kbx
./.gnupg/trustdb.gpg
./.gnome2
./.themes
...output omitted...
```

Search for files owned by group ID 1000 in the current directory.

```
[jmedinar@localhost ~]$ find -gid 1000
.
./.gnupg
./.gnupg/pubring.kbx
./.gnupg/trustdb.gpg
./.gnome2
./.themes
...output omitted...
```

The `-user`, and `-group` options can be used together to search files where file owner and group owner are different. The example below list files that are both owned by user root and affiliated with group mail.

```
[root@localhost ~]# find / -user root -group mail
/var/spool/mail
...output omitted...
```

The `-perm` option is used to look for files with a particular set of permissions. Permissions can be described as octal values, with some combination of 4, 2, and 1 for read, write, and execute.

Permissions can be preceded by a `/` or `-` sign.

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

To use a more complex example, the following command matches any file for which the user has read, write, and execute permissions, members of the group have read and write permissions, and others have read-only access:

```
[root@localhost ~]# find /home -perm 764
```

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

```
[root@localhost ~]# find /home -perm -324
```

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

```
[root@localhost ~]# find /home -perm /442
```

When used with `/` or `-`, a value of 0 works like a wild-card, since it means a permission of at least nothing.

To match any file in the /home/user directory for which others have at least read access on host, run:

```
[jmedinar@localhost ~]$ find -perm -004
```

Find all files in the /home/user directory where other has write permissions on host.

```
[jmedinar@localhost ~]$ find -perm -002
```

Searching by size

The find command can look up files that match a size specified with the `-size` option, followed by a numerical value and the unit.

find Size Units

- b - 512-byte blocks. This is the default if no unit is specified.
- c - Bytes
- w - 2-byte words
- k - Kilobytes (units of 1024 bytes)
- M - Megabytes (units of 1048576 bytes)
- G - Gigabytes (units of 1073741824 bytes)

The example below shows how to search for files with a size of 10 megabytes, rounded up.

```
[jmedinar@localhost ~]$ find -size 10M
```

To search the files with a size more than 10 gigabytes.

```
[jmedinar@localhost ~]$ find -size +10G
```

To list all files with a size less than 10 kilobytes.

```
[jmedinar@localhost ~]$ find -size -10k
```

> *IMPORTANT*
>
> *The -size option unit modifiers round everything up to single units. For example, the find -size 1M command shows files smaller than 1 MB because it rounds all files up to 1 MB.*

Searching by modification time

The `-mmin` option, followed by the time in minutes, searches for all files that had their content changed at n minutes ago in the past. The file's time-stamp is always rounded down. It also supports fractional values when used with ranges (+n and -n).

To find all files that had their file content changed 120 minutes ago on host, run:

```
[root@localhost ~]# find / -mmin 120
```

The + modifier in front of the amount of minutes looks for all files in the / that have been modified more than n minutes ago. In this example, files that were modified more than 200 minutes ago are listed.

```
[root@localhost ~]# find / -mmin +200
```

The - modifier changes the search to look for all files in the / directory which have been changed less than n minutes ago. In this example, files that were modified less than 150 minutes ago are listed.

```
[root@localhost ~]# find / -mmin -150
```

Searching by file type

The `-type` option in the find command limits the search scope to a given file type. Use the following list to pass the required flags to limit the scope of search:

- b - Block special device file
- c - Character special device file
- d - Directory
- f - Regular file
- l - Symbolic link

Search for all directories in the /etc directory on host.

```
[root@localhost ~]# find /etc -type d
/etc
/etc/tmpfiles.d
/etc/systemd
/etc/systemd/system
/etc/systemd/system/getty.target.wants
...output omitted...
```

Search for all soft links on host.

```
[root@localhost ~]# find / -type l
```

Generate a list of all block devices in the /dev directory on host:

```
[root@localhost ~]# find /dev -type b
/dev/vda1
/dev/vda
```

The `-links` option followed by a number looks for all files that have a certain hard link count. The number can be preceded by a + modifier to look for files with a count higher than the given hard link count. If the number is preceded with a - modifier, the search is limited to all files with a hard link count that is less than the given number.

Search for all regular files with more than one hard link on host:

```
1 [root@localhost ~]# find / -type f -links +1
```

More options

find supports a large number of different tests. Below is a rundown of the common ones.

Options	Description
-cmin n	Match files or directories whose content or attributes were last modified exactly n minutes ago. To specify less than n minutes ago, use -n and to specify more than n minutes ago, use +n.
-cnewer file	Match files or directories whose contents or attributes were last modified more recently than those of file.
-ctime n	Match files or directories whose contents or attributes were last modified n*24 hours ago.
-empty	Match empty files and directories.
-group name	Match file or directories belonging to group. group may be expressed as either a group name or as a numeric group ID.
-iname pattern	Like the -name test but case insensitive.
-inum n	Match files with inode number n. This is helpful for finding all the hard links to a particular inode.
-mmin n	Match files or directories whose contents were last modified n minutes ago.
-mtime n	Match files or directories whose contents were last modified n*24 hours ago.
-name pattern	Match files and directories with the specified wild-card pattern.

Options	Description
-newer file	Match files and directories whose contents were modified more recently than the specified file. This is very useful when writing shell scripts that perform file backups. Each time you make a backup, update a file (such as a log), and then use find to determine which files have changed since the last update.
-nouser	Match file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
-nogroup	Match files and directories that do not belong to a valid group.
-perm mode	Match files or directories that have permissions set to the specified mode. mode may be expressed by either octal or symbolic notation.
-samefile name	Similar to the -inum test. Matches files that share the same inode number as file name.
-size n	Match files of size n.
-type c	Match files of type c.
-user name	Match files or directories belonging to user name. The user may be expressed by user-name or by a numeric user ID.

This is not a complete list. The find man page has all the details.

Predefined actions

Let's get some work done! Having a list of results from our find command is useful, but what we really want to do is act on the items on the list. Fortunately, find allows actions to be performed based on the search results. There are a set of predefined actions and several ways to apply user-defined actions. First, let's look at a few of the predefined actions:

- -delete: Delete the currently matching file.
- -ls: List files
- -print: Output the full pathname of the matching file to standard output.
- -quit: Quit once a match has been made.

We can use find to delete files that meet certain criteria. For example, to delete files that have the file extension ".BAK", we could use this command:

```
1 [jmedinar@localhost ~]$ find ~ -type f -name '*.BAK' -delete
```

Warning: It should go without saying that you should use extreme caution when using the `-delete` action. Always test the command first by substituting the `-print` action for `-delete` to confirm the search results.

User-Defined Actions

In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the -exec action. This action works like this:

```
-exec command {} ;
```

where the command is the name of a command, `{}` is a symbolic representation of the current pathname, and the semicolon is a required delimiter indicating the end of the command. Here's an example of using `-exec` to act like the `-delete` action discussed earlier:

```
[jmedinar@localhost ~]$ find ~ -type f -name '*.BAK' -exec rm -rf {}
\;
```

notice the backslash before the semicolon is to remove the special meaning of the character.

Need more info?

🎗 Linux/Mac Terminal Tutorial: How To Use The find Command

Grep & Regular Expressions

What Are Regular Expressions?

Simply put, regular expressions are symbolic notations used to identify patterns in text. Regular expressions are supported by many command-line tools and by most programming languages to facilitate the solution of text manipulation problems.

🎗 Regular Expressions (RegEx) Tutorial

The grep command

The name `grep` is actually derived from the phrase `global regular expression print` so we can see that `grep` has something to do with regular expressions. In essence, `grep` searches text files for the occurrence of a specified regular expression and output any line containing a match to standard output.

So far, we have used `grep` with fixed strings, like so:

```
[jmedinar@localhost ~]$ ls /usr/bin | grep zip
bunzip2
bzip2
bzip2recover
funzip
...output omitted...
```

This will list all the files in the `/usr/bin` directory whose names contain the substring `zip`.

The `grep` program accepts options and arguments this way:

```
[jmedinar@localhost ~]$ grep [options] regex [file...]
```

Where regex is a regular expression.

Here is a list of the commonly used `grep` options:

- -i Ignore Case
- -v Invert match.
- -c Count the number of matches
- -l Print the name of each file that contains a match instead of the lines themselves.
- -L inverted -l option, print only the names of files that **do not** contain matches.
- -n Prefix each matching line with the number of the line within the file.
- -h Suppress the output of filenames.

To work out some examples let's start by creating some files by executing the following commands.

```
[jmedinar@localhost ~]$ cd
[jmedinar@localhost ~]$ ls /bin > dirlist-bin.txt
[jmedinar@localhost ~]$ ls /usr/bin > dirlist-usr-bin.txt
[jmedinar@localhost ~]$ ls /sbin > dirlist-sbin.txt
[jmedinar@localhost ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
```

We can perform a simple search of our list of files like this:

```
[jmedinar@localhost ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
dirlist-usr-bin.txt:bzip2
dirlist-usr-bin.txt:bzip2recover
```

and we can start adding some of our options to the mix:

```
[jmedinar@localhost ~]$ grep -l bzip dirlist-*
dirlist-bin.txt
dirlist-usr-bin.txt
```

Meta-characters and literals

While it may not seem apparent, our `grep` searches have been using regular expressions all along, albeit very simple ones. The regular expression `bzip` is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the characters `b`, `z`, `i`, and `p` are found in that order, with no other characters in between. The characters in the string `bzip` are all literal characters, in that they match themselves. In addition to literals, regular expressions may also include meta-characters that are used to specify more complex matches. Regular expression meta-characters consist of the following:

```
^ $ . [ ] { } - ? * + ( ) | \
```

All other characters are considered literals, though the backslash character is used in a few cases to create meta sequences, as well as allowing the meta-characters to be escaped and treated as literals instead of being interpreted as meta-characters.

Command line expanssions

The Bash shell has multiple ways of expanding a command line including pattern matching, home directory expansion, string expansion, and variable substitution. Perhaps the most powerful of these is the path name-matching capability, historically called globbing. The Bash globbing feature, sometimes called “wildcards”, makes managing large numbers of files easier. Using metacharacters that “expand” to match file and path names being sought, commands perform on a focused set of files at once.

The any character

The first meta-character we will look at is the dot or period character, which is used to match `any` character.

```
[jmedinar@localhost ~]$ grep -h '.zip' dirlist*.txt
```

We searched for any line in our files that matches the regular expression `.zip`

Pattern Matching

Globbing is a shell command-parsing operation that expands a wildcard pattern into a list of matching path names. Command-line metacharacters are replaced by the match list prior to command execution. Patterns that do not return matches display the original pattern request as literal text. The following are common metacharacters and pattern classes.

PATTERN	MATCHES
*	Any string of zero or more characters.
?	Any single character.
[abc...]	Any one character in the enclosed class (between the square brackets).
[!abc...]	Any one character not in the enclosed class.
[^abc...]	Any one character not in the enclosed class.
[[:alpha:]]	Any alphabetic character.
[[:lower:]]	Any lowercase character.
[[:upper:]]	Any uppercase character.
[[:alnum:]]	Any alphabetic character or digit.
[[:punct:]]	Any printable character not a space or alphanumeric.
[[:digit:]]	Any single digit from 0 to 9.
[[:space:]]	Any single white space character. This may include tabs, newlines, carriage returns, form feeds, or spaces.

Run the following commands to create some sample files.

```
1 [jmedinar@localhost ~]$ mkdir glob; cd glob
2 [jmedinar@localhost glob]$ touch alfa bravo charlie delta echo able
  baker cast dog easy
3 [jmedinar@localhost glob]$ ls
4 able  alfa  baker  bravo  cast  charlie  delta  dog  easy  echo
```

Let's run some examples will use simple pattern matches with the asterisk (*) and question mark (?) characters, and a class of characters, to match some of those file names.

```
 1 [jmedinar@localhost glob]$ ls a*
 2 able  alfa
 3 [jmedinar@localhost glob]$ ls *a*
 4 able  alfa  baker  bravo  cast  charlie  delta  easy
 5 [jmedinar@localhost glob]$ ls [ac]*
 6 able  alfa  cast  charlie
 7 [jmedinar@localhost glob]$ ls ????
 8 able  alfa  cast  easy  echo
 9 [jmedinar@localhost glob]$ ls ?????
10 baker  bravo  delta
```

Anchors

The caret ^ and dollar sign $ characters are treated as anchors in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line ^ or at the end of the line $:

```
[jmedinar@localhost ~]$ grep -h '^zip' dirlist*.txt
[jmedinar@localhost ~]$ grep -h 'zip$' dirlist*.txt
[jmedinar@localhost ~]$ grep -h '^zip$' dirlist*.txt
```

Bracket expressions and character classes

With bracket expressions, we can specify a set of characters to be matched. In this example, using a two character set:

```
[jmedinar@localhost ~]$ grep -h '[bg]zip' dirlist*.txt
```

Traditional character ranges

It's just a matter of putting all uppercase letters in a bracket expression.

```
[jmedinar@localhost ~]$ grep -h '^[A-Z]' dirlist*.txt
```

matches all filenames starting with letters and numbers:

```
[jmedinar@localhost ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

Tilde Expansion

The tilde character (~), matches the current user's home directory. If it starts a string of characters other than a slash (/), the shell will interpret the string up to that slash as a user name, if one matches, and replace the string with the absolute path to that user's home directory. If no user name matches, then an actual tilde followed by the string of characters will be used instead. In the following example the echo command is used to display the value of the tilde character. The echo command can also be used to display the values of brace and variable expansion characters, and others.

```
[jmedinar@localhost glob]$ ls ~
Desktop   Documents   Downloads   Music   Pictures   Public
Templates   tmp   Videos
```

Brace Expansion

Brace expansion is used to generate discretionary strings of characters. Braces contain a comma-separated list of strings, or a sequence expression. The result includes the text preceding or following the brace definition. Brace expansions may be nested, one inside another. Also double-dot syntax (..) expands to a sequence such that {m..p} will expand to m n o p.

```
[jmedinar@localhost glob]$ echo {Sunday,Monday,Tuesday,Wednesday}.log
Sunday.log Monday.log Tuesday.log Wednesday.log
[jmedinar@localhost glob]$ echo file{1..3}.txt
file1.txt file2.txt file3.txt
[jmedinar@localhost glob]$ echo file{a..c}.txt
filea.txt fileb.txt filec.txt
[jmedinar@localhost glob]$ echo file{a,b}{1,2}.txt
filea1.txt filea2.txt fileb1.txt fileb2.txt
[jmedinar@localhost glob]$ echo file{a{1,2},b,c}.txt
filea1.txt filea2.txt fileb.txt filec.txt
```

A practical use of brace expansion is to quickly create a number of files or directories.

```
[jmedinar@localhost glob]$ touch file{1..10}.txt
[jmedinar@localhost glob]$ ls file*
file10.txt  file1.txt  file2.txt  file3.txt  file4.txt  file5.txt
file6.txt  file7.txt  file8.txt  file9.txt
```

Variable Expansion

A variable acts like a named container that can store a value in memory. Variables make it easy to access and modify the stored data either from the command line or within a shell script.

You can assign data as a value to a variable using the following syntax:

```
[jmedinar@localhost ~]$ my_variable=some_value
```

You can use variable expansion to convert the variable name to its value on the command line. If a string starts with a dollar sign ($), then the shell will try to use the rest of that string as a variable name and replace it with whatever value the variable has.

```
[jmedinar@localhost ~]$ echo $my_variable
some_value
```

To help avoid mistakes due to other shell expansions, you can put the name of the variable in curly braces, for example ${VARIABLENAME}.

```
[jmedinar@localhost ~]$ echo ${my_variable}
some_value
```

Shell variables and ways to use them will be covered in more depth later when we get into shell scripting.

Command Substitution

Command substitution allows the output of a command to replace the command itself on the command line. Command substitution occurs when a command is enclosed in parentheses, and preceded by a dollar sign $. The $(command) form can nest multiple command expansions inside each other.

```
[jmedinar@localhost ~]$ echo Today is $(date +%A)
Today is Monday
[jmedinar@localhost ~]$ echo The time is $(date +%M) minutes past $(date +%l%p)
The time is 48 minutes past 7PM
```

Protecting Arguments from Expansion

Many characters have special meaning in the Bash shell. To keep the shell from performing shell expansions on parts of your command line, you can quote and escape characters and strings.

The backslash \ is an escape character in the Bash shell. It will protect the character immediately following it from expansion.

```
[jmedinar@localhost ~]$ echo The value of $HOME is your home directory.
The value of /home/jmedinar is your home directory.
[jmedinar@localhost ~]$ echo The value of \$HOME is your home directory.
The value of $HOME is your home directory.
```

In the preceding example, protecting the dollar sign from expansion caused Bash to treat it as a regular character and it did not perform variable expansion on $HOME.

To protect longer character strings, single quotes (') or double quotes (") are used to enclose strings. They have slightly different effects. Single quotes stop **all** shell expansion. Double quotes stop **most** shell expansion.

Use double quotation marks to suppress globbing and shell expansion, but still allow command and variable substitution.

```
[jmedinar@localhost ~]$ myhost=$(hostname -s); echo $myhost
localhost
[jmedinar@localhost ~]$ echo "***** hostname is ${myhost} *****"
***** hostname is localhost *****
```

Use single quotation marks to interpret all text literally.

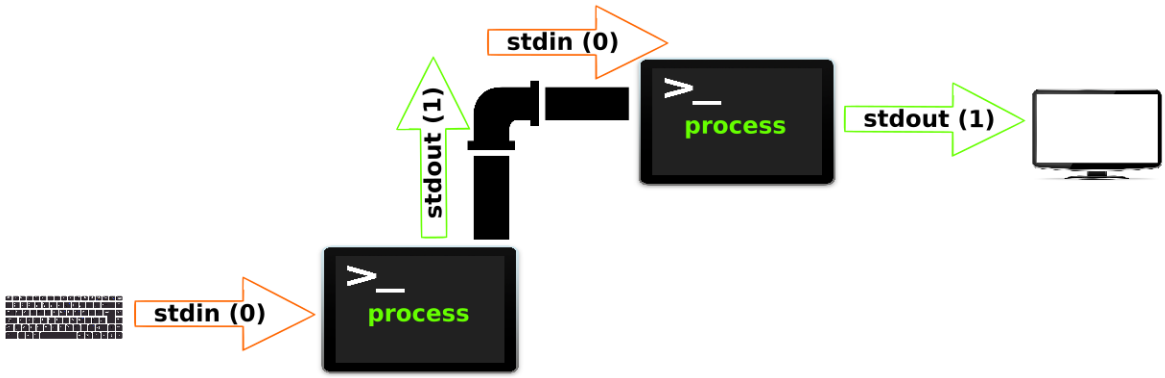
```
[jmedinar@localhost ~]$ echo "Will variable $myhost evaluate to $(hostname -s)?"
Will variable localhost evaluate to localhost?
[jmedinar@localhost ~]$ echo 'Will variable $myhost evaluate to $(hostname -s)?'
Will variable $myhost evaluate to $(hostname -s)?
```

Not enough information?

🎞 The Grep Command

Pipes

A pipeline is a sequence of one or more commands separated by the pipe character (|). A pipe connects the standard output of the first command to the standard input of the next command.



Pipelines allow the output of a process to be manipulated and formatted by other processes before it is output to the terminal. One useful mental image is to imagine that data is "flowing" through the pipeline from one process to another, being altered slightly by each command in the pipeline through which it flows.

The ability of commands to read data from standard input and send it to standard output is utilized by a shell feature called pipelines. Using the pipe operator | (vertical bar), the standard output of one command can be piped into the standard input of another:

```
1 command1 | command2
```

Pipeline Examples

This example takes the output of the ls command and uses less to display it on the terminal one screen at a time.

```
1 [jmedinar@localhost ~]$ ls -l /usr/bin | less
```

The output of the ls command is piped to wc -l, which counts the number of lines received from ls and prints that to the terminal.

```
1 [jmedinar@localhost ~]$ ls | wc -l
```

In this pipeline, head will output the first 10 lines of output from ls -t, with the final result redirected to a file.

```
[jmedinar@localhost ~]$ ls -t | head -n 10 > /tmp/ten-last-changed-files
```

We have been using this feature already but let's learn a bit more about it. In the following example we are using the command `ls` to list the files in a directory and then **piping** the resulting output to a `grep` command and looking for the regular expression `zip`

```
[jmedinar@localhost ~]$ ls /usr/bin | grep zip
```

We can do super complex and strange sequences of commands with pipe and that is why is one of the most powerful tools we can learn. let's consider the following example:

We want to see if any of our file-systems its using more than 20% of space

1. We can see the disk utilization with the command `df`

```
[jmedinar@localhost ~]$ df
Filesystem                              1K-blocks     Used Available Use% Mounted on
devtmpfs                                  8021052        0   8021052   0% /dev
tmpfs                                     8041784   506232   7535552   7% /dev/shm
tmpfs                                     8041784     1868   8039916   1% /run
/dev/mapper/fedora_localhost--live-root  71724152 22211960  45825792  33% /
tmpfs                                     8041784       88   8041696   1% /tmp
/dev/sda1                                  999320   273604    656904  30% /boot
/dev/mapper/fedora_localhost--live-home 398404080 34988920 343107604  10% /home
```

But we only want to see those on 10% of Use

```
[jmedinar@localhost ~]$ df | grep '1[0-9]%'
/dev/mapper/fedora_localhost--live-home 398404080 34989852 343106672  10% /home
```

Nice!. but now we only need to print the exact percentage value!

```
[jmedinar@localhost ~]$ df | grep '1[0-9]%' | awk '{print $5}'
10%
```

and we want to remove the '%' symbol!

```
[jmedinar@localhost ~]$ df | grep '1[0-9]%' | awk '{print $5}' | sed 's/%//g'
10
```

Do not worry for now about the `awk` and `sed` commands. See how we can construct more complex commands and get more accurate results by knowing how to manage our `finds`, `greps`, `pipes` and `regular expressions`!

Re-directions

Let's unleash what may be one of the coolest features of the command line. It's called I/O redirection. The "I/O" stands for input/output and with this facility, you can redirect the input and output of commands to and from files. To show off this facility,
we will introduce the following commands:

- `cat` - Concatenate files
- `sort` - Sort lines of text
- `uniq` - Report or omit repeated lines
- `grep` - Print lines matching a pattern
- `wc` - Print newline, word, and byte counts for each file
- `head` - Output the first part of a file
- `tail` - Output the last part of a file
- `tee` - Read from standard input and write to standard output and files

Standard input, output, and error

Many of the programs that we have used so far produce an output and receive input of some kind.

This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

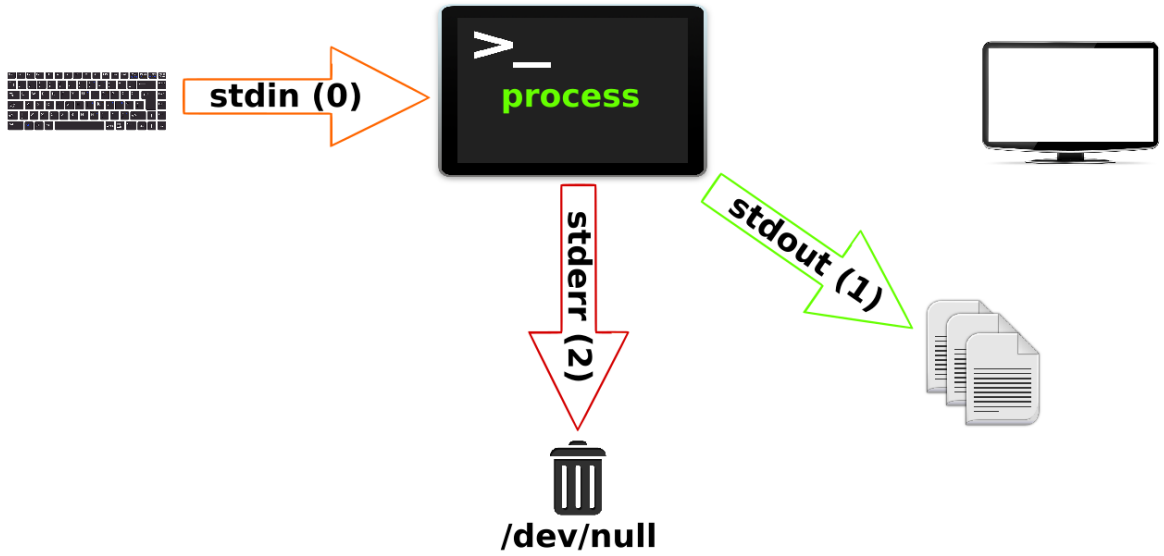
Remember in Linux "everything is a file" programs such as `ls` actually send their results to a **special file called standard output** or **stdout** and their status messages to another file called **standard error or stderr**. By default, both standard **stdout** and stderr are **linked** to the **screen** and not saved into a disk file. In addition, many programs take input from a facility called **standard input** or **stdin** which is, by default, attached to the **keyboard**.

NUMBER	CHANNEL NAME	DESCRIPTION	DEFAULT CONNECTION	USAGE
0	**stdin**	Standard input	Keyboard	read only

NUMBER	CHANNEL NAME	DESCRIPTION	DEFAULT CONNECTION	USAGE
1	**stdout**	Standard output	Terminal	write only
2	**stderr**	Standard error	Terminal	write only

Redirections

I/O redirection changes how the process gets its input or output. Instead of getting input from the keyboard, or sending output and errors to the terminal, the process reads from or writes to files. Redirection lets you save messages to a file that are normally sent to the terminal window. Alternatively, you can use redirection to discard output or errors, so they are not displayed on the terminal or saved.



Redirecting **stdout** suppresses process output from appearing on the terminal. As seen in the following table, redirecting only **stdout** does not suppress **stderr** error messages from displaying on the terminal. If the file does not exist, it will be created. If the file does exist and the redirection is not one that appends to the file, the file's contents will be overwritten.

If you want to discard messages, the special file `/dev/null` quietly discards channel output redirected to it and is always an empty file.

USAGE	EXPLANATION
> file	redirect stdout to **overwrite** a file
>> file	redirect stdout to **append** to a file
2> file	redirect stderr to overwrite a file

USAGE	EXPLANATION
2> /dev/null	discard stderr error messages by redirecting to /dev/null
> file 2>&1	redirect stdout and stderr to overwrite the same file
>> file 2>&1	redirect stdout and stderr to append to the same file

Redirecting standard output

I/O redirection allows us to redefine where **stdout** goes. To redirect **stdout** to another file instead of the screen, we use the **">"** redirection operator followed by the name of the file.

```
1 [jmedinar@localhost ~]$ ls -l /usr/bin > ls-output.txt
```

what happens if you do the following?

```
1 [jmedinar@localhost ~]$ > ls-output.txt
```

Appending standard output

We append redirected output to a file instead of overwriting the file from the beginning by using the **">>"** redirection operator, like so

```
1 [jmedinar@localhost ~]$ ls -l /etc/ >> ls-output.txt
```

Using the ">>" operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the ">" operator had been used.

Redirecting standard error

Redirecting **stderr** lacks the ease of a dedicated redirection operator. To redirect standard error we must refer to its **file descriptor**. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output, and error, the shell references them internally as file descriptors 0, 1 and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
1 [jmedinar@localhost ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor "2" is placed immediately before the redirection operator to perform the redirection of the standard error to the file `ls-error.txt`.

Redirecting standard output and standard error to one file

There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, the traditional way, which works with old versions of the shell:

```
[jmedinar@localhost ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First, we redirect standard output to the file `ls-output.txt` and then we redirect file descriptor 2 (standard error) to file descriptor one (standard output) using the notation `2>&1`

Recent versions of bash provide a second, more streamlined method for performing this

```
[jmedinar@localhost ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation **"&>"** to redirect both standard output and standard error to the file `ls-output.txt`. You may also append the standard output and standard error streams to a single file like so:

```
[jmedinar@localhost ~]$ ls -l /bin/usr &>> ls-output.txt
```

Disposing Of unwanted output

Sometimes we don't want output from a command, we just want to throw it away. This applies, particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called "/dev/null". This file is a system device called a bit bucket that accepts input and does nothing with it. To suppress error messages from a command, we do this:

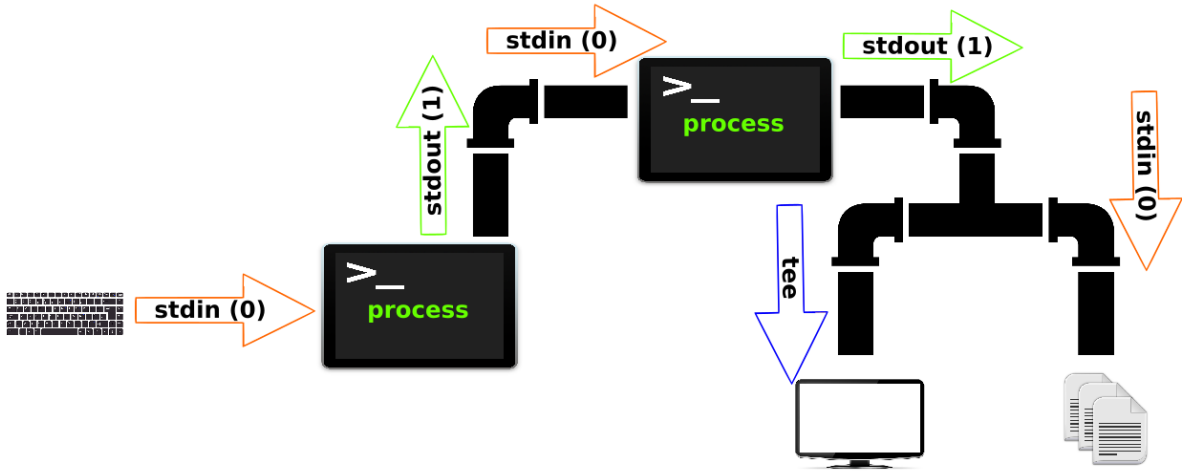
```
[jmedinar@localhost ~]$ ls -l /bin/usr 2> /dev/null
```

Pipelines, redirection, and the tee command

When redirection is combined with a pipeline, the shell sets up the entire pipeline first, then it redirects input/output. If output redirection is used in the middle of a pipeline, the output will go to the file and not to the next command in the pipeline. In this example, the output of the ls command goes to the file, and less displays nothing on the terminal.

```
[jmedinar@localhost ~]$ ls > /tmp/saved-output | less
```

The tee command overcomes this limitation. In a pipeline, tee copies its standard input to its standard output and also redirects its standard output to the files named as arguments to the command. If you imagine data as water flowing through a pipeline, tee can be visualized as a "T" joint in the pipe which directs output in two directions.



Pipeline Examples Using the tee Command

This example redirects the output of the ls command to the file and passes it to less to be displayed on the terminal one screen at a time.

```
[jmedinar@localhost ~]$ ls -l | tee /tmp/saved-output | less
```

If tee is used at the end of a pipeline, then the final output of a command can be saved and output to the terminal at the same time.

```
[jmedinar@localhost ~]$ ls -t | head -n 10 | tee /tmp/ten-last-changed-files
```

cat – concatenate files

The cat command reads one or more files and copies them to standard output or in other words displays the content of a file like so:

```
$ cat ls-output.txt
```

sort - order the output

The sort command arranges the output lines first in alphabetical order but we can also request other types.

```
$ ls /bin /usr/bin | sort
```

uniq - report or omit repeated lines

The `uniq` command is often used in conjunction with sort. `uniq` accepts a sorted list of data from either standard input or a single filename argument and, by default, removes any duplicates from the list.

```
$ ls /bin /usr/bin | sort | uniq
```

wc – word count/ lines and bytes

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
$ ls /bin /usr/bin | sort | uniq | wc -l
```

head / tail – print first / last part of files

Sometimes you don't want all the output from a command. You may only want the first few lines or the last few lines. By default, both commands print 10 lines of text, but this can be adjusted with the "-n" option:

```
$ head -n 5 ls-output.txt tail -n 5 ls-output.txt
```

The command tail has an option that allows you to view files in real-time. This is useful for watching the progress of log files as they are being written.

```
$ tail -f /var/log/messages
```