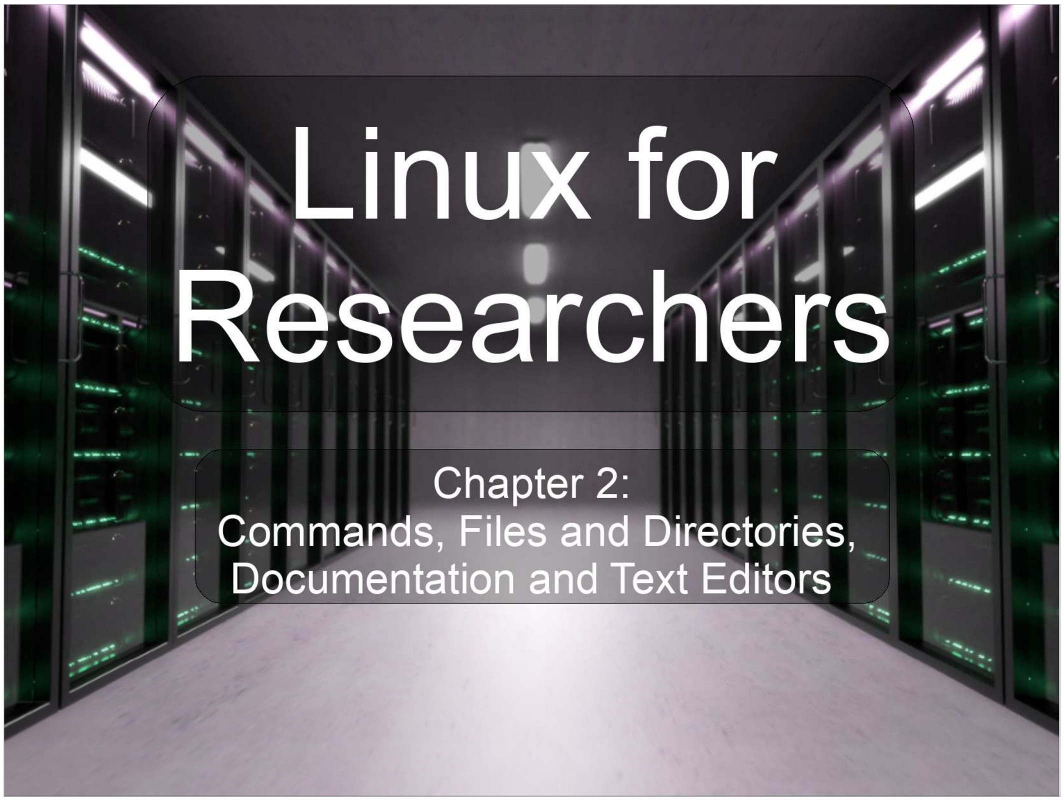


Today we'll start talking about things that are specific to Linux and other unix-like operating systems.

We'll come back to these concepts again and again, so this is just a quick first pass.

Part1: The Command Line Shell

This is a teletype terminal of the type I used back in the 1970s, when all we had was the command line, and We Liked IT!

Linux is still maybe a little more weighted toward the command line than Windows. Looking at the Linux World, you could say that almost anything you can do at the command line can also be done graphically. In the Windows world, I think it's fair to say that almost anything you can do grapically can also be done from the command line.

The two worlds are converging: Linux's graphical interfaces are continually improving, and Windows keeps improving its command-line interface.

We can all appreciate the value of a graphical interface. It's intuitive (if it's well-designed) and its “discoverable”, in that you can browse around a graphical program's menus and find out what the program can do. But what's the value of a command-line interface? If graphical interfaces are good, why do all major operating systems continue to improve their command-line interfaces?

The Command Line:

Prompt Command Results

```
~/demo> ls
clus.pdf      data-for-everybody.1.dat  phase2
cluster.pdf   ForYourEyesOnly.dat       readme.txt
cpuinfo.dat   phase1                    ReadMe.txt
```

Output of the “ls” command, which lists the files in the current directory.

Why should you do things from the command line?:

* In Linux, graphical tools provide a front-end to help you do tasks, but you can do more from the command line.

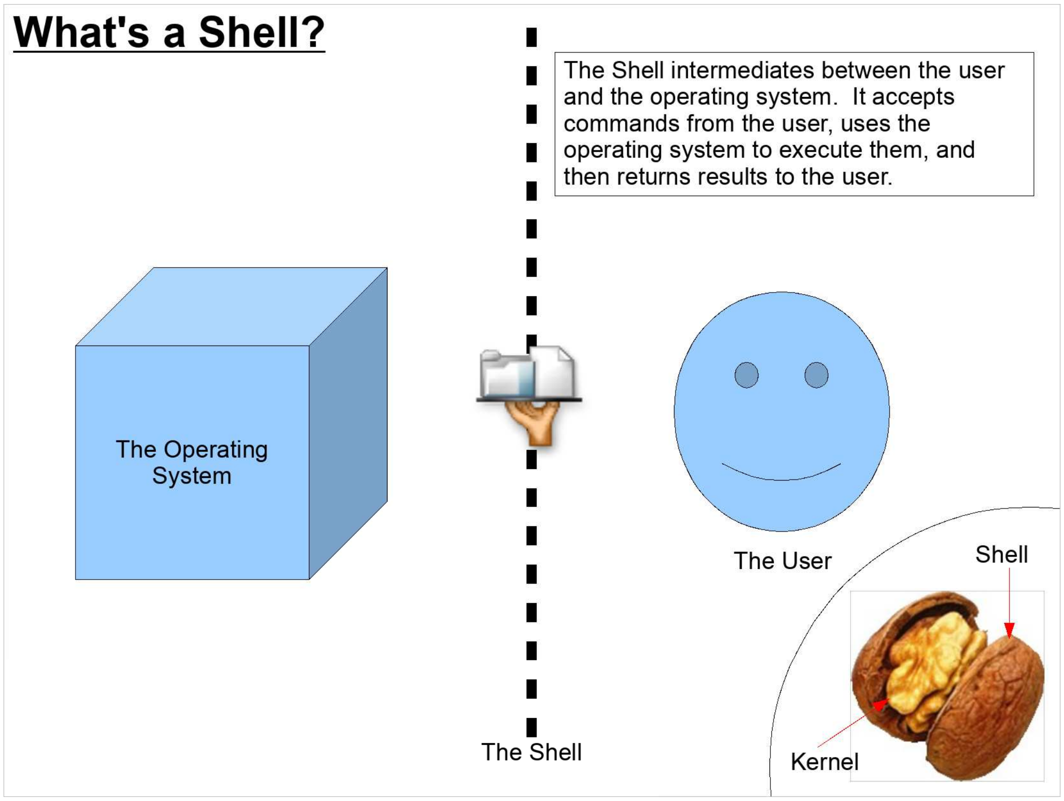
* There are several sets of graphical tools available for Linux, so if you learn one of them you may find that it's not available on the next computer you use.

* There's no guarantee that a given computer will have graphical tools installed, or even a monitor.

* Text commands are easily reproduced. It's easy to document what you've done, or to tell someone else how to do it, or to automate what you've done.

The answer is that the command line has its own advantage. Here are some of the things that might make someone choose to use the command line under Linux.

The last item is the most important, I think. This is true for all operating systems, not just Linux, and it's why all major operating systems still have a command-line interface and continue to improve it.



There are graphical shells and command-line shells. When you log in to Microsoft Windows, you're using a graphical shell. In that case, you communicate with the shell by pointing and clicking. Today we'll be talking about command-line shells, where you communicate by typing commands.

The diagram in the corner explains why we call it a “shell”. It's the interface between the inner “kernel” of the operating system and the outer world.

Command-line Shells:

Command-line shells accept typed commands, parse them and execute them. They also:

• Expand wild-card expressions.

• Usually store a history of previously-typed commands, and provide a way of recalling these.

• Provide a set of built-in functions that supplement (or sometimes replace) the commands provided by the operating system.

• Provide the user with the ability to define abbreviations for commands (aliases).

• Maintain a set of user-defined variables that can be used in command lines (environment variables and shell variables).

....and.....

Command-line Shells (cont'd):

....

• Often have the ability to auto-complete partially typed commands or file names.

• Provide control structures (if/then/else, while/do) that allow the user to write programs in the shell's language (shell scripts).

• Provide the plumbing to connect commands together with pipes and to redirect the input and output of commands.

The two most commonly used shells used on Linux systems are bash and tcsh, which behave somewhat differently. Bash is Richard Stallman's re-implementation of the original Bourne Shell (sh), which he named the “Bourne Again Shell”. Tcsh is an enhanced version of csh, which was an early alternative to the Bourne Shell.

A Few Useful Linux Commands:

ls	List the contents of a directory.
pwd	Show the name of the current directory.
cd	Change the current directory.
less or more	Show the contents of a file, one page at a time.
cp	Copy a file.
mv	Move (rename, relocate or both) a file.
rm	Delete (remove) a file.
mkdir	Make a new directory.
rmdir	Delete (remove) a directory.
man	Show docs (manual pages) for a command.
ln	Make a link to a file.
cat	Spit out the concatenated contents of one or more files, without paging.
touch	Change the timestamp on a file, or create an empty file.
which	Find a command in the search path.

This is just a list to get you started.

As you can see, the commands are typically terse.

Command Syntax:

```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Linux commands are often modified by the addition of switches or qualifiers like the “-l”, for “long”, switch used in the ls command above. These modifiers will often take one of these forms:

* A dash followed by a letter or number, optionally followed by an argument
* Two dashes followed by a word, optionally followed by an argument.

For ls, some useful switches are:

-l Gives more information about the files.
-T Combined with -l, sorts the files in reverse time order.
-S Combined with -l, sorts the files in order of descending size.
-a Lists all files, including hidden files.

Multiple single-letter switches can often be combined, like “ls -lT” instead of “ls -l -T”

In this case, we can change the behavior of the “ls” command by adding the “-l” switch.

Note, though, that Linux commands were all developed independently, and they have a long history. Syntax conventions have evolved over time, and different developers have used different conventions.

We'll see examples of some odd command syntax with commands like “tar” and “ps”.

Command-Line History:

```
~/demo> history
  349  16:14   wget http://download.adobe.com/pub/adobe/magic/svgviewer....
  350  16:15   tar tzvf adobesvg-3.01x88-linux-i386.tar.gz
  351  16:15   tar xzvf adobesvg-3.01x88-linux-i386.tar.gz
  352  16:15   cd adobesvg-3.01
  353  16:15   dir
  354  16:15   cd ..
  355  16:15   rm adobesvg-3.01*
  356  16:15   rm -rf adobesvg-3.01*
  357  16:19   git clone git://people.freedesktop.org/~cworth/svg2pdf
  358  16:19   cd svg2pdf
  359  16:19   dif
  360  16:19   dir
  361  16:19   git pull
  362  16:19   make
  363  16:19   dir
  364  16:20   ./svg2pdf ../drawing.svg
  365  16:20   ./svg2pdf ../drawing.svg junk.pdf
  366  16:20   acroread junk.pdf
```

The “history” command shows you commands you've recently entered.

You can use the up and down arrow keys to recall previously-typed commands and re-use them. If you know the beginning of a previously-entered command, you can re-run it by entering a “!” followed by the beginning of the command.

The PATH Environment Variable:

```
~/demo> echo $PATH
.:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

The PATH variable defines a search path for the shell to use when looking for a program. It's composed of a list of directory names, separated by colons. When looking for a program, the shell starts at the left of the list and looks in each directory until it finds a program with the matching name (or fails).

```
~/demo> which ls
/bin/ls
```

The “which” command looks through the search path, just as the shell would, and tells you where the shell would find a given program.

With the PATH shown above, a local administrator can add local programs to /usr/local/bin to make them available to users.

Note that:

1. “.”, the current directory, is included. This is not generally the case for users with administrative privileges, for security reasons.

2. By putting an alternative program with the same name in /usr/local/bin, a local administrator can provide a modified version of a program that overrides any version that might already exist in /bin or /usr/bin.

Modifying the PATH Variable:

In the bash shell:

```
~/demo> echo $SHELL
/bin/bash
~/demo> export PATH="/home/bryan/bin:$PATH"
```

In the tcsh shell:

```
~/demo> echo $SHELL
/bin/tcsh
~/demo> setenv PATH "/home/bryan/bin:$PATH"
```

Aliases and Shell Built-in Commands:

```
~/demo> which echo
echo: shell built-in command.

~/demo> which rm
rm:      aliased to rm -i
```

Creating aliases:

In the bash shell:

```
~/demo> alias blarg=ls
```

In the tcsh shell:

```
~/demo> alias blarg ls
```

```
~/demo> blarg
clus.pdf     data-for-everybody.1.dat  phase2
cluster.pdf  ForYourEyesOnly.dat       readme.txt
cpuinfo.dat  phase1                    ReadMe.txt
```

Shell Startup Files:

Both bash and tcsh read a set of startup files when the user logs in. These files can be used to automatically set environment variables (like "PATH"), define aliases, or execute other shell commands.

For bash:
Add commands to the file ".bash_profile", in your home directory.

For tcsh:
Add commands to the file ".login", in your home directory.

Each of the shells actually looks at different startup files under different circumstances, but the files above are a good place to start.

Plugging Commands Together With Pipes (|):

Linux commands are modular, and can be plugged together easily to do complex things. The output of one command can be sent to the input of another, and so on. (We'll see more of this when we start talking about shell scripts.) There are two common ways of doing this, “pipes” and “backticks”:

```
~/demo/phase1> ls -l | less  (shows output of “ls” one page at a time)

~/demo/phase2> ls -l
total 16
drwxr-x--- 2 bkw1a bkw1a 4096 Jan 19 10:41 .
drwxr-x--- 4 bkw1a bkw1a 4096 Jan 19 10:39 ..
-rw-r--r-- 1 bkw1a bkw1a    0 Jan 19 10:41 even_more_junk.txt
-rw-r--r-- 1 bkw1a bkw1a    0 Jan 19 10:41 junk1.txt
-rw-r--r-- 1 bkw1a bkw1a    0 Jan 19 10:41 junk2.txt
-rw-r--r-- 1 bkw1a bkw1a   32 Jan 19 10:38 more_stars.txt
-rw-r--r-- 1 bkw1a bkw1a   18 Jan 19 10:38 some_stars.txt

~/demo/phase2> ls -l | grep stars | sed -e 's/star/STAR/' | awk '{print
$3,$NF}'

bkw1a more_STARS.txt
bkw1a some_STARS.txt
```

Here we've introduced the “less” command, which will show you its input one page at a time. You can use it to view the contents of a file one page at a time by typing, e.g., “less file.txt” or “cat file.txt | less”.

Less is the GNU project's successor to the “more” command found in the original Unix. On most Linux systems, “more” is an alternative name for “less”.

We've also mentioned the grep, sed and awk commands, which we'll talk more about when we discuss scripting. For the example above, grep selects only certain lines from its input, sed modifies its input in a specified way, and awk selects only certain columns of its input.

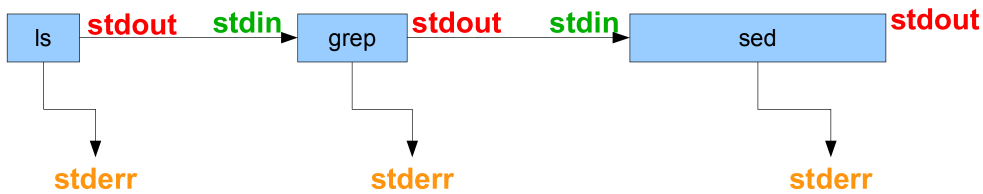
Stdin, Stdout, and Stderr:

stdout Channel through which output is printed by a program.

stdin Channel from which a program obtains input data.

stderr Channel through which a program reports errors.

```
ls -l | grep stars | sed -e 's/st/ST/'
```



Stdin, stdout and stderr are the connectors through which commands are plumbed together.

You can think of each command as a little device that has one input (stdin) and two outputs (stdout and stderr). By plugging outputs and inputs together, you can build up a long pipeline of commands that work together to do a complex task.

Redirecting Output to a File:

```
ls -l | grep stars > output.dat
```

Redirecting stdout into a file

```
ls -l | grep stars >> output.dat
```

Appending lines to an existing file

Redirecting Both Stdout and Stderr to a file:

Under tcsh:

```
ls -l | grep stars >& output.dat
```

Under bash:

```
ls -l | grep stars > output.dat 2>&1
```

Plugging Commands Together With Backticks (`):

```
~/demo/phase2> ls

bad_users.txt    laundry_list.txt   recipes.txt
good_users.txt   random_junk.txt    ugly_users.txt
...et cetera.

~/demo/phase2> ls | grep users

bad_users.txt
good_users.txt
ugly_users.txt

~/demo/phase2> grep Bryan `ls | grep users`

ugly_users.txt:Bryan
```

Commands between backticks are evaluated, then their output is inserted into the command line just as though you'd typed it there yourself, directly.

Here we see another way to use the grep command. If it's given a list of filenames as arguments, it will operate on the content of those files instead of on its input. (In this mode it prints the name of the file in addition to the matching line.)

This is typical of many Linux commands. If given an input file name, they'll work on that file. Otherwise they'll wait for input to be piped into them.

Part 2: Files and Directories:

Listing the Files in the Current Directory:

```
~/demo> ls
clus.pdf     data-for-everybody.1.dat  phase2
cluster.pdf  ForYourEyesOnly.dat       readme.txt
cpuinfo.dat  phase1                    ReadMe.txt

~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

The first thing a user will probably want to do is look around. We can do this with the “ls” command. Note that “ls” and “ls -l” do different things.

Let's spend some time dissecting the output of the ls -l command.

Case-sensitive File Names:

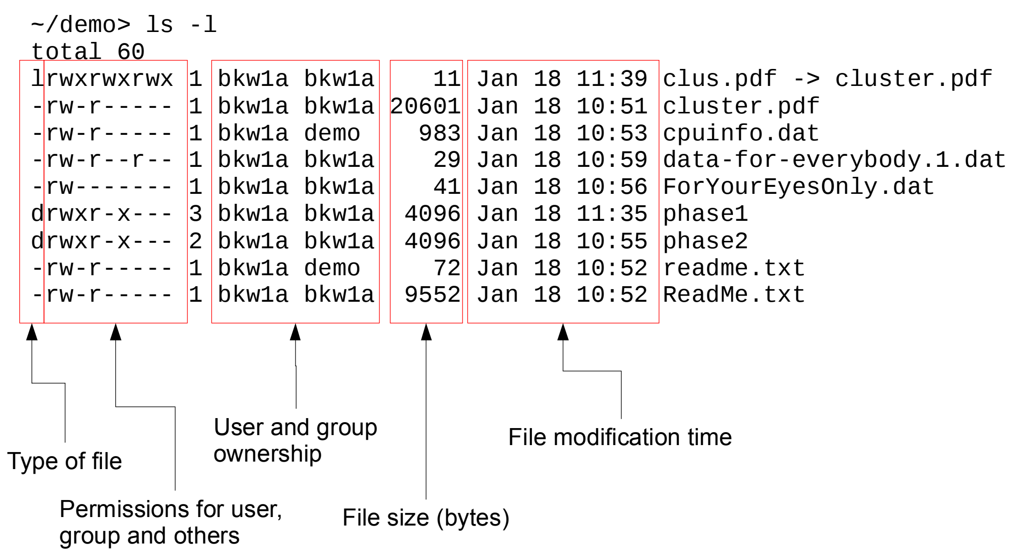
```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Under Linux, files are typically case-sensitive. This means that “readme.txt” is a completely different file from “ReadMe.txt”. This is unlike Windows or OS X, where a filename's case is preserved, but ignored.

We see that Linux file names are usually case-sensitive.

This is a feature of the filesystem, not the operating system, per se. We'll talk about filesystems at another time. Some filesystems used under Linux are not case-sensitive, but the most common ones are.

Understanding the Output of the “ls” Command:



Here are some other features of the output of our “ls” command.

For now, we'll ignore the third column. This is the number of “hard links”. For files, it will almost always be 1. For directories, it will be 2 + the number of subdirectories inside the directory. We'll talk about “why” later.

The “file size” colum reports what you'd expect for files. For directories, though, it reports the the size of the directory <u>excluding its contents</u>. For a directory, this number is the size of all of the “metadata” associated with this directory: the file names, permissions, *et cetera*. This isn't generally very useful.

Symbolic Links:

```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Symbolic links are like alternative names for a file or directory. In the example above, “clus.pdf” is a symbolic link pointing to a real file called “cluster.pdf”. You can have as many symbolic links as you like pointing to a given real file. (Symbolic links can even point to other symbolic links, but there's a limit on how many levels of this you can do.)

To create a symbolic link, use the “ln” command:

```
ln -s RealFile SymlinkFile
```

Be careful of the order! It's easy to get the file names reversed.

Symbolic links can be a wonderful way to solve otherwise thorny problems. Many Unix administrators, when asked what's the most useful Unix command, will say “ln -s”, the command used for creating symbolic links.

Directory files:

```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Directories can contain other directories, and these subdirectories show up in the output of "ls -l". In the example above, "phase1" and "phase2" are two subdirectories of the directory we're looking at. Each of these subdirectories may contain its own files and more subdirectories.

The “Current Directory”:

You can see what directory you're currently working in by using the “pwd” command:

```
~/demo> pwd
/home/bryan/demo
```

Note that the path to a file or directory is given as a list of parent directories, separated by slashes, starting with the root directory (“/”). In this case, the current working directory is “/home/bryan/demo”.

You can change your current directory by using the “cd” command, like:

```
~/demo> cd phase1
```

Or, equivalently:

```
~/demo> cd /home/bryan/demo/phase1
```

In the first case, we specify the name of a directory relative to the current directory, and in the second case we explicitly give the full path name (the complete name of the directory we're interested in.)

Let's pause for a minute to look at the way directories are laid out on a typical Linux system.

First, there's the concept of a “current directory”.

The "Home Directory":

Each user has a "home directory". This directory will be your current directory right after you log in.

```
~/demo> echo $HOME
/home/bryan
```

You can use the $HOME environment variable in commands, to refer to your home directory.

```
~/demo> ls $HOME/demo
```

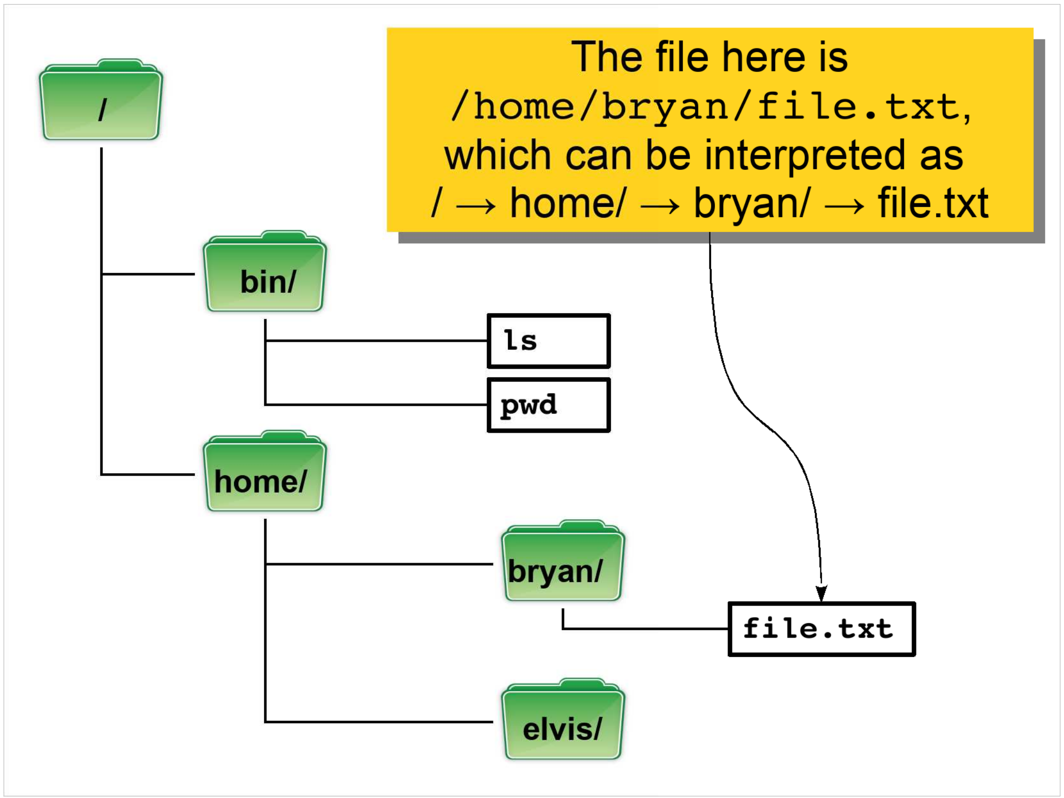
You can also refer to your home directory as "~", in most shells.

```
~/demo> ls ~/demo
```

If you just type "cd" by itself, it will take you to your home directory.

$HOME is an "environment variable". These are similar to the variables used in computer programs. In fact, you can write programs in the shell language, too. These are usually called "scripts" or "shell scripts". They can be used to automate shell tasks you do often.

The $HOME variable is set for you automatically when you log into the computer. We'll talk later about how modify the values of environment variables, or create new ones of your own.

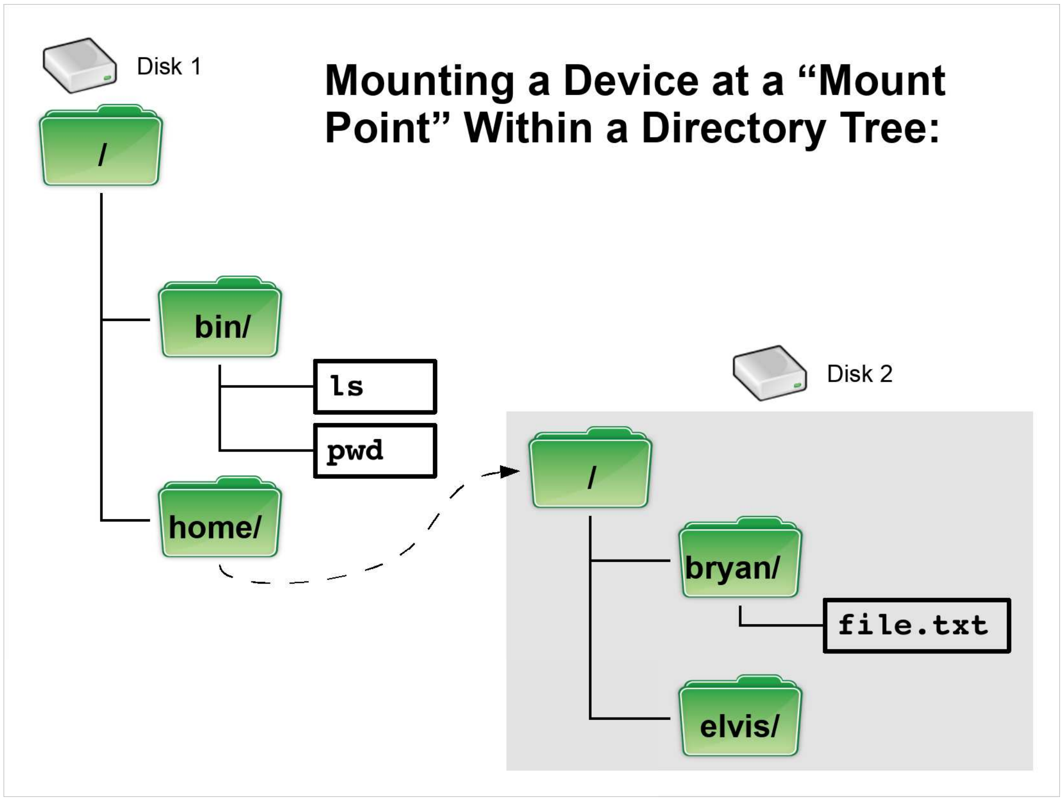


Here's a graphical representation of a highly simplified Linux directory tree. One of the basic principles of Linux (and other varieties of Unix) is that there's only one directory tree. Everything lives somewhere under the “/” (root) directory.

This is unlike Windows, for example, where each device has a separate directory tree. In Windows we have a directory tree on drive C:, a different one on drive D:, and so on. As we'll see a little later, under Linux all files on all devices show up somewhere in the same directory tree, with “/” at its top.

Note that, whereas Windows uses “\” as the directory separator, Linux uses “/”.

To find out how much space is used by all the files underneath a given directory, you can use the “du” command, like “du -s -k phase1”, which would tell you the total size of all files under the subdirectory “phase1”, in kilobytes.



The directory tree of each physical device is grafted onto the same tree, with the root directory ("/") at the top. There are no "C:" or "D:" drives under Linux. Every file you have access to lives in the same tree, and you don't need to care what device the file lives on.

Viewing Mounted Filesystems with “df”:

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                      73545144  37268984  32479988  54% /
/dev/sda2               101105     45519     50365  48% /boot
/dev/sdb1            721075720 630461080  53986040  93% /data
tmpfs                  2008536         0   2008536   0% /dev/shm
home.private:/home   721075744 621413088  63034048  91% /home
mail.private:/var/spool/mail
                     721075744 621413088  63034048  91% /var/spool/mail
```

For now, just notice the last column of this output, which shows several different filesystems mounted within the directory tree. We'll talk about the details of this when we discuss filesystems, in a later meeting.

The Linux Directory Tree:

From the Linux Filesystem Hierarchy Standard: http://proton.pathname.com/fhs/

/	Top directory of the entire file system hierarchy.
/bin	Essential programs that need to always be availale for all users.
/boot	Boot loader files.
/dev	Special files representing various devices.
/etc	System-wide configuration files specific to this computer.
/home	Users' home directories.
/lib	Libraries essential for the binaries in /bin/ and /sbin/.
/media	Mount points for removable media such as CD-ROMs.
/mnt	Temporarily mounted filesystems.
/opt	Optional application software packages.
/proc	Virtual filesystem, documenting kernel and process status as text files.
/root	Home directory for the root user.
/sbin	Essential system programs.
/srv	Site-specific data which is served by the system.
/tmp	Temporary files (see also /var/tmp).
/usr	Secondary tree, containing the majority of user utilities and applications.
/var	Tertiary tree for local data, specific to this computer.

Two important principles in Linux are:
1. There's only one directory tree.
2. Everything is a file.

Although it's possible to arrange the directories on a Linux system in many ways, there's an evolving standard layout called the Linux Filesystem Hierarchy Standard (fhs). All major Linux distributors use this now.

File Types ("Everything is a File"):

Regular files (-):

```
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
```

Directories/folders (d):

```
drwxr-x---   3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
```

Symbolic Links (l):

```
lrwxrwxrwx 1 bkw1a bkw1a 11 Jan 18 11:39 clus.pdf -> cluster.pdf
```

Block or Character Device Special Files (b or c):

```
~/demo> ls -l /dev/sda1
brw-r----- 1 root disk 8, 1 Dec 26 10:23 /dev/sda1
~/demo> ls -l /dev/ttyS0
crw-rw---- 1 root uucp 4, 64 Dec 26 10:22 /dev/ttyS0
```

Unix Domain Sockets or Named Pipes (s or p):

```
~/demo> ls -l /var/lib/mysql/mysql.sock
srwxrwxrwx 1 mysql mysql 0 Jul  1  2008 /var/lib/mysql/mysql.sock
~/demo> ls -l /var/run/xdmctl/xdmctl
prw--w---- 1 root root 0 Dec 26 10:24 /var/run/xdmctl/xdmctl
```

For now, we'll only talk about regular files, symbolic links and directories. We'll save the other types for another time.

File Permissions:

```
 rwxrwxrwx
-rwxr----- 1 bkw1a demo 20601 Jan 18 10:51 myprogram
```

User Group

U G O

For each file, three sets of permission bits can be set, referring to three different classes of people: The user who owns the file (u), the group that owns the file (g) and everybody else ("other", or o). For each of these classes, several permission bits can be set (or not), including read (r), write(w) and execute (x).

For the file above, the user (bkw1a) has permission to read, write or execute the file. The owning group (demo) has permission to read the file, but not to write it or execute it. Other users have no permission to do anything with the file.

Note that permissions are interpreted as though they were read from right to left. For example, if the user permissions give no write access, but the "other" permissions grant write access, then the user still won't be able to write the file, even though others can. This is true since the user has been explicitly denied access in the user permissions.

Looking back at the output of "ls", let's examine the file permissions column.

There are other things besides permissions that control access to files. We'll talk about attributes, ACLs and other things later.

File Timestamps:

mtime: The file's “modificaton time”. This is the time that the file's contents were last modified. This is the time that “ls” displays by default.

ctime: Somewhat confusingly, this is the “change time”. This is the time at which the file's properties (excluding contents) were last changed. For example, if the file's permissions are changed, or its ownership is changed, the ctime will change.

To see the ctime of files, use “ls --time=ctime”

atime: This is the file's “access time”, showing the last time the file was looked at. Many administrators currently disable the updating of atime stamps, since they entail some I/O overhead and are seen as being of little value.

To see the atime of files, use “ls --time=atime”

Typically, Linux filesystems store several different timestamps for each file.

Ctime is useful because hackers will often modify the mtime stamps of any files they've changed, to hide the hackers' activities. The almost always forget to change the ctime stamp, though.

“Hidden” Files:

```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
~/demo> ls -al
total 100
drwxr-x---   4 bkw1a bkw1a  4096 Jan 18 11:39 .
drwxr-x--x 269 bkw1a bkw1a 28672 Jan 18 18:45 ..
-rw-r-----   1 bkw1a bkw1a    45 Jan 18 10:55 .anotherexample
lrwxrwxrwx   1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r-----   1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r-----   1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r--   1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw-r-----   1 bkw1a bkw1a    34 Jan 18 10:54 .example1
-rw-------   1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x---   3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x---   2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r-----   1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r-----   1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Another switch to use with “ls”. Look at the difference between the output of the two commands.

Any file whose name begins with a dot is a “hidden” file. These aren't hidden for any security reason, they're just not shown so things will be less cluttered. Generally, these files contain configuration information of one type or another.

Two particular hidden files will always appear in every directory. These are “.” and “..”. These files give you a way to refer to the current directory (.) and its parent directory (..). For example, the command “ls .” does the same thing as “ls”. To move up one directory with cd, you could type “cd ..”. The properties of the “.” and “..” directories, as shown by ls, tell about permissions, ownership, etc. of the respective directories.

Note that the root directory also contains a “..” entry. In this case, since there's nothing above the root directory, the “..” entry just points back to root itself. (“/” is its own parent.)

Wild-cards ("globbing"):

The commands "ls -l junk.txt" or "ls /home/bryan/demo/phase1" will tell us about the named file or directory. We can also use wild-cards to specify filenames that match a pattern. This process is called "globbing" in Linux:

*	Match any string of characters.
?	Match any single character.
[abc]	Match a single a, b or c.
[a-zA-Z]	Match any character in the range a-z or A-Z.

Note that these patterns won't match filenames beginning with a dot ("hidden" files).

It's important to remember how wild-cards work on the Linux command line. If we type a command like "ls *.txt", here's what happens internally:

"ls *.txt" is expanded to read "ls a.txt b.txt c.txt...." by inserting the names of any files that match. Then the resulting expanded command line is executed.

The "ls" command itself never sees the wild-card characters. The command-line interpreter (the "shell") expands the wild-card expression before invoking "ls". Because of this, wild-cards work for any command, since wild-card support doesn't have to be built into the command itself. Commands just get this functionality for free. This is another example of the modular philosophy behind the design of Linux and other varieties of Unix.

There are two types of pattern-matching in Linux, "globbing" and "regular expressions". For most things you'll do on the command-line, globbing is used. We'll talk about regular expressions, which are much more powerful but also more difficult to understand, when we discuss scripting.

Some Examples of Wild-card Matching:

```
~/demo> ls -l
total 2
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bkw1a  bkw1a       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bkw1a  bkw1a       758 Jul 30 09:02 test1.txt

~/demo> ls -l t?.sh
-rw-rw-r--    1 bkw1a  bkw1a       466 Aug  6 17:48 t2.sh

~/demo> ls -l [ab]*
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 b.1

~/demo> ls -l [a-c]*
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bkw1a  bkw1a         0 Aug  6 18:42 c.1
```

Here are a few examples of glob-style wild-card matching.

Part 3: Documentation

Documentation: Command-line help:

Many commands will tell you about themselves if you give them a "-h" or "--help" switch on the command line. For example:

```
~/demo> ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print octal escapes for nongraphic characters
      --block-size=SIZE      use SIZE-byte blocks
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information)
                               with -l: show ctime and sort by name
                               otherwise: sort by ctime
  -C                         list entries by columns
      --color[=WHEN]         control whether color is used to distinguish file
                               types.  WHEN may be `never', `always', or `auto'
  -d, --directory            list directory entries instead of contents,
                               and do not dereference symbolic links
  -D, --dired                generate output designed for Emacs' dired mode
  -f                         do not sort, enable -aU, disable -lst
  -F, --classify             append indicator (one of */=>@|) to entries
.....
```

Note that this is just a convention, and not all commands will honor it. As we noted before, these commands have a long history, and were written by many authors.

Documentation: Man Pages:

"Man Pages" (online documents in a standard format) are available for most common commands. The "man" command will show these to you, one page at a time. To exit from man, type "q" (for "quit"). To go to the next page, press the spacebar. To go back up, press "b".

```
~/demo> 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 alphabetically if none of -cftuvSUX nor --sort.

       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 ..

q=quit,b=back,space=forward,h=help
```

For information about using the man command, don't hesitate type type "man man".

Man pages are the most common type of online documentation for Unix-like operating systems.

Documentation: Info Pages:

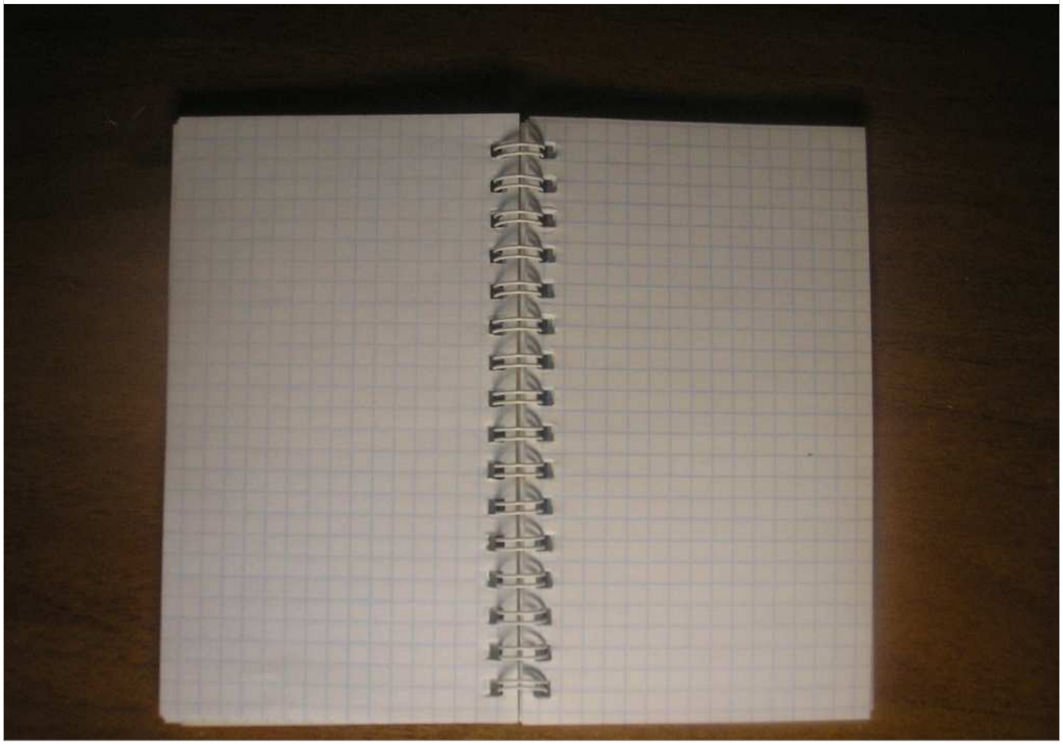
"GNU Info Pages" are another standard format for online documentation. Fewer commands have info pages, but when present this documentation may be more extensive than the command's man page. Info pages are arranged in a tree, with links between documents, much like a primitive version of the World Wide Web.

```
~/demo> info ls
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directo\
ry listing

10.1 `ls': List directory contents
==================================

The `ls' program lists information about files (of any type, including
directories).  Options and file arguments can be intermixed
arbitrarily, as usual.

   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the file name.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.

   By default, the output is sorted alphabetically, according to the
locale settings in effect.(1) If standard output is a terminal, the
output is in columns (sorted vertically) and control characters are
output as question marks; otherwise, the output is listed one per line
and control characters are output as-is.
--zz-Info: (coreutils.info.gz)ls invocation, 54 lines --Top---------------------
Welcome to Info version 4.8. Type ? for help, m for menu item.
```

Some commands have only info pages. These commands will typically have a minimal man page that only refers you to the info page.

For information about navigating around inside info, try typing "info info" at the command line.

Part 4: Text Editors



Text Files vs. Word Processor Files:

If I open a text editor, type the line “This is a test” and save the file as “file.txt”, the file will have the following data inside it:

file.txt

```
This is a test.
```

If I open a word processor, type the line “This is a test” and save the file as “file.doc”, the data inside the file will look like this:

file.doc

```
��Q ^Z�@^@^@^@^@^@^@^@^@^@^@^@^@^@^@;^@^C^@��     ^@^F^@^@^@^@^@^@^@^@^$
^@^@^@^K^@^@^@^L^@^@^@^M^@^@^@^N^@^@^@^O^@^@^@^P^@^@^@^Q^@^@^@^R^@^@^@^S^@^@^@^$
^@^@^@^K^@^@^@^L^@^@^@^M^@^@^@^N^@^@^@^O^@^@^@^P^@^@^@^Q^@^@^@^R^@^@^@^S^@^@^@^$
^@^@����^F      ^B^@^@^@^@�@^@^@^@^@F^X^@^@^@Microsoft Word-Document^@
^@^@^@MSWordDoc^@^P^@^@^@Word.Document.8^@��q^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^$
^@^A^@[^@^O^@^B^@^@^@^@^@^@^@\^@^@^P��^B^@\^@^@^@^G^@D^@e^@f^@a^@u^@l^@t^@^@^@^$
^@^P^@^S�^@^@^T�x^@^@^@ ^@/^P^A^A^R^A ^@^@^@^D^@L^@i^@s^@t^@^@^@^B^@^Q^@^D^@^J^$
^@^@^@^@����^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^$
^@^@^@^@^@^@^@^@^@^@^Q^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^$
^@^@^@T^@^@^@^K^@^@^@`^@^@^@^L^@^@^@l^@^@^@^M^@^@^@x^@^@^@^Q^@^@^@\204^@^@^@^B^$
6^_B^V^_B^V^_86^CB^A^G^A^B^BB^A^N^A/�^A:^A^[^BB^A8^A6^CB^A^\^BB^@^E%^O^Q4B^@^B$
B^@"$^@?B^Y^@^R,^@4^Z)^]B4^K% 3^EB^X-^Z2)^K2B^\^@^X^R:^BB^A^E^A^B^DB^B"^@^S^G^@$
^N/B^V/B^V/A^Q^CB^A!^AA^BB^A.^A^O�^A^R^A^[^BB^A^A^A^Q^EB^@^P/^[^_$^ABA^AB^O^[($
1!B^O^[(^FB^A9^FB^AA^A1^BB^@^D1!^R+^EB^A^R^A+^BB^A^V^A/^B^A^AB^B^A^A2^A^K^BB^@^$
0^@^XB<^@^FA^@/^DB^A,^B^^^@^O=^]^K^FB^R<^H^[#^RB^F^@^G^@^DB^A5^A^K^FB^@^C^_^@^T$
^N^OB1^LA^^^A1^M^CB^A4^A-^BB^A^W^A^^^BB^@^S1!^R^A4%^A   >+^LBA^^^A1^M1A^@^BB^A^$
^N^OB1^LA^^^A1^M^FB^@^@^EB^A^@^B^C^A^D^D^C^A^D^C^C^@^C^D^C^D^@^C^C^A^D^B^C^A^@^$
^E^_B^H,^P^]B^O^N^CB^A^G^A^U^BB^A7^A.^BB^A^^^A^X^CB^@^EA^A$^Y"^@^BB^@^G^P^]B^O^$
^E^_B^H,^P^]B^O^N^FB^@^@^EB^A^@^B^C^A^D^H^C^@^C^D^C^D^@^F^C^A^@^BB^@^F^M^^&^P^^$
B^@^_;3@^F^@9"^^^B:*^S$^G;B<^O^F#^R^M^V*        ^M B"^V.^@^B2^BB^@^F8^@^K<^O?^B$
^Y1^M B^Q^HA^[^O^DB^B^Q^@^H^Y^QB^G+^A.^U^EB^@   ^H^T>^F^W^E9>^H^@^BB^B^Q^@^N^Y^$
...etc.
```

It's important to know how to use a text editor because most Linux configuration files are text files. For example, the “.login” or “.bash_profile” shell startup files are just plain text files.

Some Common Linux Text Editors:

vi	The original Unix “visual editor”. Found on all Unix-like computers. Rather non-intuitive.
emacs	Developed later by RMS, emacs is a very powerful, extensible editor. Emacs and vi are the two most commonly-used Unix editors.
pico	A small, intuitive editor included with the “pine” mail program. Found on systems where pine is installed.
nano	A standalone clone of pico, with enhancements.

I recommend you learn to use emacs, but any of these are fine. It may be easiest to start out by learning pico/nano.

Using nano:

```
~/demo> nano purplecow.txt
```

```
  GNU nano 2.2.4              File: purplecow.txt                      Modified

I've never seen a purple cow.
I never hope to see one.
But I can tell you anyhow,
I'd rather see than be one.

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Instructions at the bottom show you how to do basic operations, like saving the file and exiting nano. Note that “^” is just shorthand for “hold down the CTRL key”.

Some Emacs Commands:

Starting emacs

> `emacs` ENTER
to start emacs.
> `emacs filename` ENTER
to start emacs and load a file
> `emacs -nw filename` ENTER
to start emacs with no new window (load file)

Quitting

CTRL **X** CTRL **C**
quit emacs (Can be used with impunity -- the system will prompt if the workspace has not yet been saved.)
CTRL **G**
aborts any command in progress

Help

CTRL **H T**
to see the tutorial
CTRL **H A** ***topic*** ENTER
to see help about *topic*
CTRL **X U**
undo the last command

Cursor Positioning

CTRL **F** or →
forward (right) one character.
CTRL **B** or ←
back (left) one character
CTRL **P** or Up
up one character
CTRL **N** or ↓
down one character
ESC **B**
left one word
ESC **F**
right one word

CTRL **A**
to beginning of line
CTRL **E**
to end of line
ESC **<**
start of document
ESC **>**
end of document
ESC **V**
page up
CTRL **V**
page down

CTRL **L**
cursor in middle of screen
CTRL **U** ***20*** CTRL **N**
advance *20* lines
CTRL **X W**
display the line number where the cursor is located
ESC **X** ***goto-line*** ENTER ***999*** ENTER
go to line number *999*

Search and Replace

CTRL **S** ***patterntext*** ENTER
search for *patterntext*; cursor moves as you type. Press ENTER once at the correct location
CTRL **R** ***patterntext*** ENTER
search backwards for *patterntext*; cursor moves as you type.
CTRL **S** ENTER ENTER
search for the next occurrence
ESC **%** ***oldstring*** ENTER ***newstring*** ENTER
Search for *oldstring* and replace it with *newstring*. The **Y** key confirms each replacement, **N** skips it, **Q** to exit

Regions

CTRL **SPC**
set mark at cursor
CTRL **W**
kill region
ESC **W**
copy region to kill ring
CTRL **Y**
yank back last thing killed

Loading and Saving

CTRL **X** CTRL **F** ***filename***
create new *filename* for editing (clears workspace)
CTRL **X** CTRL **F** ***filename***
load in *filename* for editing
CTRL **X** CTRL **W** ***filename***
write (save) as *filename*
CTRL **X** CTRL **S**
resave under the current filename (based on above or name given when starting `emacs`)

Emacs cheat sheet:http://ccrma.stanford.edu/guides/package/emacs/emacs.html

This is copied from the Stanford site, above.

Thanks!