16.1. Basic Commands

The first commands a novice learns

ls

The basic file "list" command. It is all too easy to underestimate the power of this humble command. For example, using the `-R`, recursive option, **ls** provides a tree-like listing of a directory structure. Other useful options are `-S`, sort listing by file size, `-t`, sort by file modification time, `-v`, sort by (numerical) version numbers embedded in the filenames, [1] `-b`, show escape characters, and `-i`, show file inodes (see Example 16-4).

```
bash$ ls -l
-rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt


bash$ ls -lv
 total 0
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
```

The *ls* command returns a non-zero exit status when attempting to list a non-existent file.

```
bash$ ls abc
ls: abc: No such file or directory


bash$ echo $?
2
```

Example 16-1. Using *ls* to create a table of contents for burning a CDR disk

```
#!/bin/bash
# ex40.sh (burn-cd.sh)
# Script to automate burning a CDR.


SPEED=10         # May use higher speed if your hardware supports it.
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
# DEVICE=/dev/cdrom     For older versions of cdrecord
DEVICE="1,0,0"
DEFAULTDIR=/opt  # This is the directory containing the data to be burned.
                 # Make sure it exists.
                 # Exercise: Add a test for this.

# Uses Joerg Schilling's "cdrecord" package:
# http://www.fokus.fhg.de/usr/schilling/cdrecord.html

#  If this script invoked as an ordinary user, may need to suid cdrecord
#+ chmod u+s /usr/bin/cdrecord, as root.
#  Of course, this creates a security hole, though a relatively minor one.

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Default directory, if not specified on command-line.
else
    IMAGE_DIRECTORY=$1
fi

# Create a "table of contents" file.
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# The "l" option gives a "long" file listing.
# The "R" option makes the listing recursive.
# The "F" option marks the file types (directories get a trailing /).
echo "Creating table of contents."

# Create an image file preparatory to burning it onto the CDR.
mkisofs -r -o $IMAGEFILE $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image ($IMAGEFILE)."

# Burn the CDR.
echo "Burning the disk."
echo "Please be patient, this will take a while."
wodim -v -isosize dev=$DEVICE $IMAGEFILE
#  In newer Linux distros, the "wodim" utility assumes the
#+ functionality of "cdrecord."
exitcode=$?
echo "Exit code = $exitcode"

exit $exitcode
```

cat, **tac**

cat, an acronym for *concatenate*, lists a file to `stdout`. When combined with redirection (> or >>), it is commonly used to concatenate files.

```
# Uses of 'cat'
cat filename                          # Lists the file.

cat file.1 file.2 file.3 > file.123   # Combines three files into one.
```

The `-n` option to **cat** inserts consecutive numbers before all lines of the target file(s). The `-b` option numbers only the non-blank lines. The `-v` option echoes nonprintable characters, using ^ notation. The `-s` option squeezes multiple consecutive blank lines into a single blank line.

See also Example 16-28 and Example 16-24.

In a pipe, it may be more efficient to redirect the `stdin` to a file, rather than to **cat** the file.

```
cat filename | tr a-z A-Z

tr a-z A-Z < filename   #  Same effect, but starts one less process,
                        #+ and also dispenses with the pipe.
```

tac, is the inverse of *cat*, listing a file backwards from its end.

rev

reverses each line of a file, and outputs to `stdout`. This does not have the same effect as **tac**, as it preserves the order of the lines, but flips each one around (mirror image).

```
bash$ cat file1.txt
This is line 1.
 This is line 2.


bash$ tac file1.txt
This is line 2.
 This is line 1.


bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cp

This is the file copy command. **`cp file1 file2`** copies `file1` to `file2`, overwriting `file2` if it already exists (see Example 16-6).

Particularly useful are the `-a` archive flag (for copying an entire directory tree), the `-u` update flag (which prevents overwriting identically-named newer files), and the `-r` and `-R` recursive flags.

```
cp -u source_dir/* dest_dir
#  "Synchronize" dest_dir to source_dir
#+  by copying over all newer and not previously existing files.
```

mv

This is the file *move* command. It is equivalent to a combination of **cp** and **rm**. It may be used to move multiple files to a directory, or even to rename a directory. For some examples of using **mv** in a script, see Example 10-11 and Example A-2.

When used in a non-interactive script, **mv** takes the `-f` (*force*) option to bypass user input.

When a directory is moved to a preexisting directory, it becomes a subdirectory of the destination directory.

```
bash$ mv source_directory target_directory

bash$ ls -lF target_directory
total 1
 drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 source_directory/
```

rm

Delete (remove) a file or files. The `-f` option forces removal of even readonly files, and is useful for bypassing user input in a script.

The *rm* command will, by itself, fail to remove filenames beginning with a dash. Why? Because *rm* sees a dash-prefixed filename as an *option*.

```
bash$ rm -badname
rm: invalid option -- b
 Try `rm --help' for more information.
```

One clever workaround is to precede the filename with a " -- " (the *end-of-options* flag).

```
bash$ rm -- -badname
```

Another method to is to preface the filename to be removed with a `dot-slash` .

```
bash$ rm ./-badname
```

When used with the recursive flag `-r`, this command removes files all the way down the directory tree from the current directory. A careless **rm -rf *** can wipe out a big chunk of a directory structure.

rmdir

Remove directory. The directory must be empty of all files -- including "invisible" *dotfiles* [2] -- for this command to succeed.

mkdir

Make directory, creates a new directory. For example, **`mkdir -p project/programs/December`** creates the named directory. The *-p* option automatically creates any necessary parent directories.

chmod

Changes the attributes of an existing file or directory (see Example 15-14).

```
chmod +x filename
# Makes "filename" executable for all users.

chmod u+s filename
# Sets "suid" bit on "filename" permissions.
# An ordinary user may execute "filename" with same privileges as the file's owner.
# (This does not apply to shell scripts.)


chmod 644 filename
#  Makes "filename" readable/writable to owner, readable to others
#+ (octal mode).

chmod 444 filename
#  Makes "filename" read-only for all.
#  Modifying the file (for example, with a text editor)
#+ not allowed for a user who does not own the file (except for root),
#+ and even the file owner must force a file-save
#+ if she modifies the file.
#  Same restrictions apply for deleting the file.


chmod 1777 directory-name
#  Gives everyone read, write, and execute permission in directory,
#+ however also sets the "sticky bit".
#  This means that only the owner of the directory,
#+ owner of the file, and, of course, root
#+ can delete any particular file in that directory.

chmod 111 directory-name
#  Gives everyone execute-only permission in a directory.
#  This means that you can execute and READ the files in that directory
#+ (execute permission necessarily includes read permission
#+ because you can't execute a file without being able to read it).
#  But you can't list the files or search for them with the "find" command.
#  These restrictions do not apply to root.

chmod 000 directory-name
#  No permissions at all for that directory.
#  Can't read, write, or execute files in it.
#  Can't even list files in it or "cd" to it.
#  But, you can rename (mv) the directory
#+ or delete it (rmdir) if it is empty.
#  You can even symlink to files in the directory,
#+ but you can't read, write, or execute the symlinks.
#  These restrictions do not apply to root.
```

chattr

Change file **attr**ibutes. This is analogous to **chmod** above, but with different options and a different invocation syntax, and it works only on *ext2/ext3* filesystems.

One particularly interesting **chattr** option is `i`. A **chattr +i `filename`** marks the file as immutable. The file cannot be modified, linked to, or deleted, *not even by root*. This file attribute can be set or removed only by *root*. In a similar fashion, the `a` option marks the file as append only.

```
root# chattr +i file1.txt


root# rm file1.txt

rm: remove write-protected regular file `file1.txt'? y
 rm: cannot remove `file1.txt': Operation not permitted
```

If a file has the `s` (secure) attribute set, then when it is deleted its block is overwritten with binary zeroes. [3]

If a file has the `u` (undelete) attribute set, then when it is deleted, its contents can still be retrieved (undeleted).

If a file has the `c` (compress) attribute set, then it will automatically be compressed on writes to disk, and uncompressed on reads.

The file attributes set with **chattr** do not show in a file listing (**ls -l**).

ln

Creates links to pre-existings files. A "link" is a reference to a file, an alternate name for it. The **ln** command permits referencing the linked file by more than one name and is a superior alternative to aliasing (see Example 4-6).

The **ln** creates only a reference, a pointer to the file only a few bytes in size.

The **ln** command is most often used with the `-s`, symbolic or "soft" link flag. Advantages of using the `-s` flag are that it permits linking across file systems or to directories.

The syntax of the command is a bit tricky. For example: **`ln -s oldfile newfile`** links the previously existing `oldfile` to the newly created link, `newfile`.

If a file named `newfile` has previously existed, an error message will result.

Which type of link to use?

As John Macdonald explains it:

Both of these [types of links] provide a certain measure of dual reference -- if you edit the contents of the file using any name, your changes will affect both the original name and either a hard or soft new name. The differences between them occurs when you work at a higher level. The advantage of a hard link is that the new name is totally independent of the old name -- if you remove or rename the old name, that does not affect the hard link, which continues to point to the data while it would leave a soft link hanging pointing to the old name which is no longer there. The advantage of a soft link is that it can refer to a different file system (since it is just a reference to a file name, not to actual data). And, unlike a hard link, a symbolic link can refer to a directory.

Links give the ability to invoke a script (or any other type of executable) with multiple names, and having that script behave according to how it was invoked.

Example 16-2. Hello or Good-bye

```
#!/bin/bash
# hello.sh: Saying "hello" or "goodbye"
#+          depending on how script is invoked.

# Make a link in current working directory ($PWD) to this script:
#    ln -s hello.sh goodbye
# Now, try invoking this script both ways:
# ./hello.sh
# ./goodbye


HELLO_CALL=65
GOODBYE_CALL=66

if [ $0 = "./goodbye" ]
then
  echo "Good-bye!"
  # Some other goodbye-type commands, as appropriate.
  exit $GOODBYE_CALL
fi

echo "Hello!"
# Some other hello-type commands, as appropriate.
exit $HELLO_CALL
```

man, **info**

These commands access the manual and information pages on system commands and installed utilities. When available, the *info* pages usually contain more detailed descriptions than do the *man* pages.

There have been various attempts at "automating" the writing of *man pages*. For a script that makes a tentative first step in that direction, see Example A-39.

Notes

[1] The `-v` option also orders the sort by *upper- and lowercase prefixed* filenames.

[2] *Dotfiles* are files whose names begin with a *dot*, such as `~/.Xdefaults`. Such filenames do not appear in a normal **ls** listing (although an **ls -a** will show them), and they cannot be deleted by an accidental **rm -rf ***. Dotfiles are generally used as setup and configuration files in a user's home directory.

[3] This particular feature may not yet be implemented in the version of the ext2/ext3 filesystem installed on your system. Check the documentation for your Linux distro.