COMP 350
Introduction to DevOps Tools and Techniques

Lecture 3
Manipulating Files, Links, Commands, IO Redirection

Hakan Ayral

Manipulating Files and Directories

- cp—Copy files and directories.
- mv—Move/rename files and directories.
- mkdir—Create directories.
- rm—Remove files and directories.
- ln—Create hard and symbolic links.

mkdir—Create Directories

- The mkdir command is used to create directories. It works like this:

```
mkdir directory
mkdir directory1 directory2 ...
```

- `mkdir dir` → will create a single directory named *dir*
- `mkdir dir1 dir2 dir3` → will create three directories named *dir1*, *dir2*, and *dir3*

cp—Copy Files and Directories

- The ***cp*** command copies files or directories. It can be used two different ways:
  - to copy the single file or directory *itemSrc* to file or directory *itemDst* and:

```
cp itemSrc itemDst
```

  - to copy multiple items (either files or directories) into a directory.

```
cp item directory
```

```
cp item1 item2 item3… directory
```

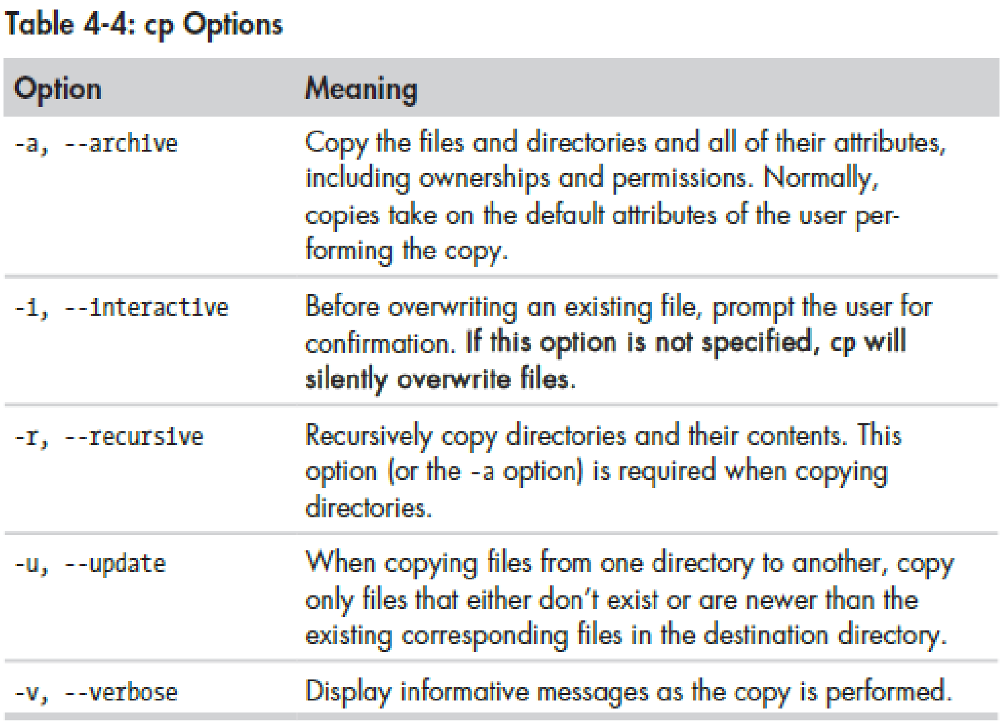


Table 4-4: cp Options

Option	Meaning
`-a, --archive`	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, cp will silently overwrite files.**
`-r, --recursive`	Recursively copy directories and their contents. This option (or the -a option) is required when copying directories.
`-u, --update`	When copying files from one directory to another, copy only files that either don't exist or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the copy is performed.

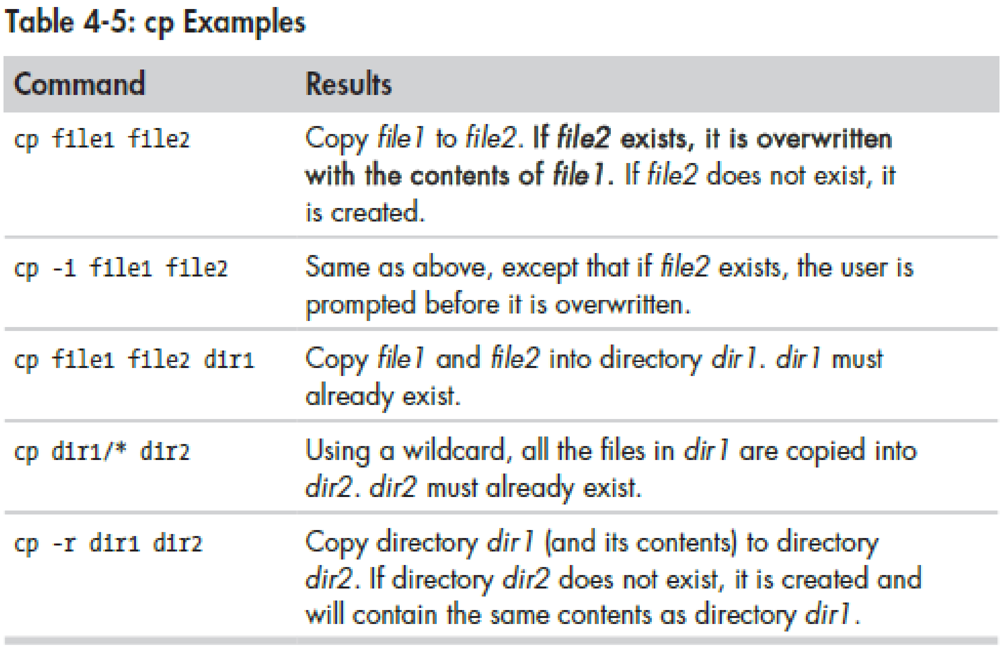


Table 4-5: cp Examples

Command	Results
`cp file1 file2`	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created.
`cp -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`cp file1 file2 dir1`	Copy *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, all the files in *dir1* are copied into *dir2*. *dir2* must already exist.
`cp -r dir1 dir2`	Copy directory *dir1* (and its contents) to directory *dir2*. If directory *dir2* does not exist, it is created and will contain the same contents as directory *dir1*.

mv—Move and Rename Files

- The ***mv*** command performs both file moving **<u>and file renaming</u>**, depending on how it is used.
- In either case, <u>the original filename no longer exists after the operation</u>.
- mv is used in much the same way as cp, to move or rename file or directory *itemSrc* to *itemDst:*

```
mv itemSrc itemDst
```

- to move one or more items from one directory to another:

```
mv item directory
mv item1 item2 item3... directory
```

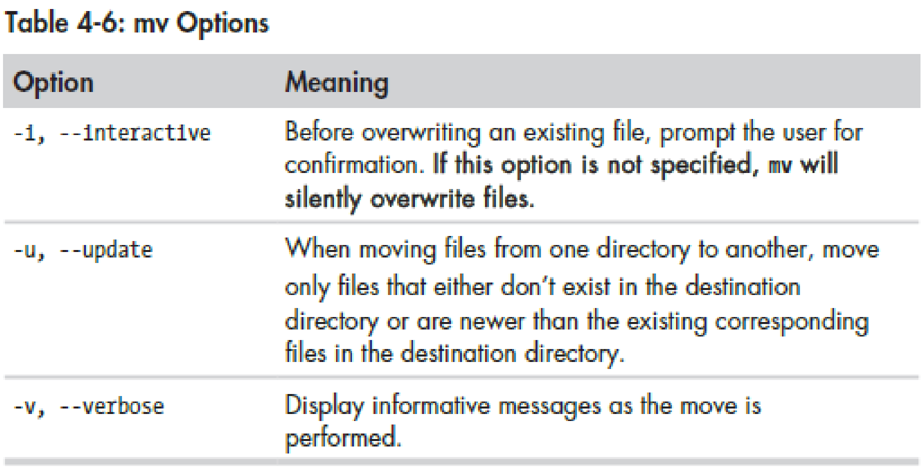


Table 4-6: mv Options

Option	Meaning
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. If this option is not specified, mv will silently overwrite files.
`-u, --update`	When moving files from one directory to another, move only files that either don't exist in the destination directory or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the move is performed.

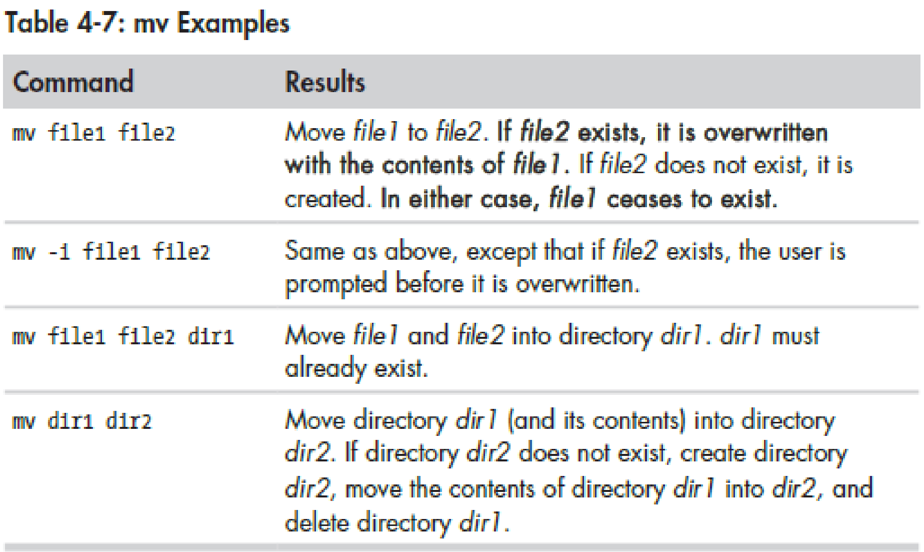


Table 4-7: mv Examples

Command	Results
`mv file1 file2`	Move *file1* to *file2*. If *file2* exists, it is overwritten with the contents of *file1*. If *file2* does not exist, it is created. In either case, *file1* ceases to exist.
`mv -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`mv file1 file2 dir1`	Move *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`mv dir1 dir2`	Move directory *dir1* (and its contents) into directory *dir2*. If directory *dir2* does not exist, create directory *dir2*, move the contents of directory *dir1* into *dir2*, and delete directory *dir1*.

rm—Remove Files and Directories

- The ***rm*** command is used to remove (delete) files and directories, like this:

```
rm item
rm item1 item2 item3...
```

- where *item* is the name of one or more files or directories.

BE CAREFUL WITH RM!

- **Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with rm, it's gone.** Linux assumes you're smart and you know what you're doing.
- Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:

```
rm *.html
```

which is correct, but if you accidentally place a space between the * and the .html like so:

```
rm * .html
```

the rm command will delete all the files in the directory and then complain that there is no file called *.html*.

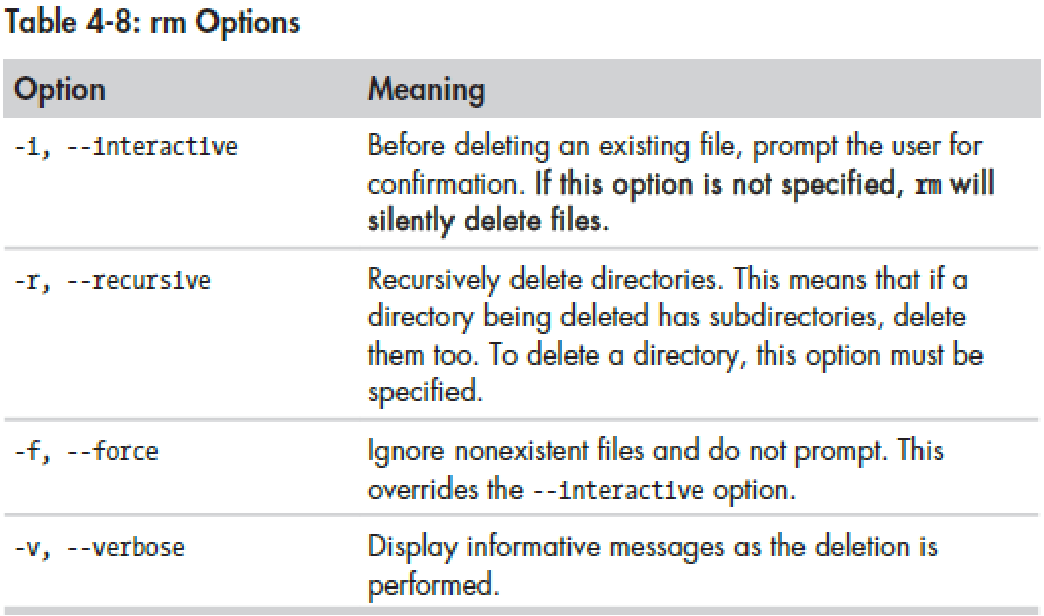


Table 4-8: rm Options

Option	Meaning
`-i, --interactive`	Before deleting an existing file, prompt the user for confirmation. **If this option is not specified, `rm` will silently delete files.**
`-r, --recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f, --force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v, --verbose`	Display informative messages as the deletion is performed.

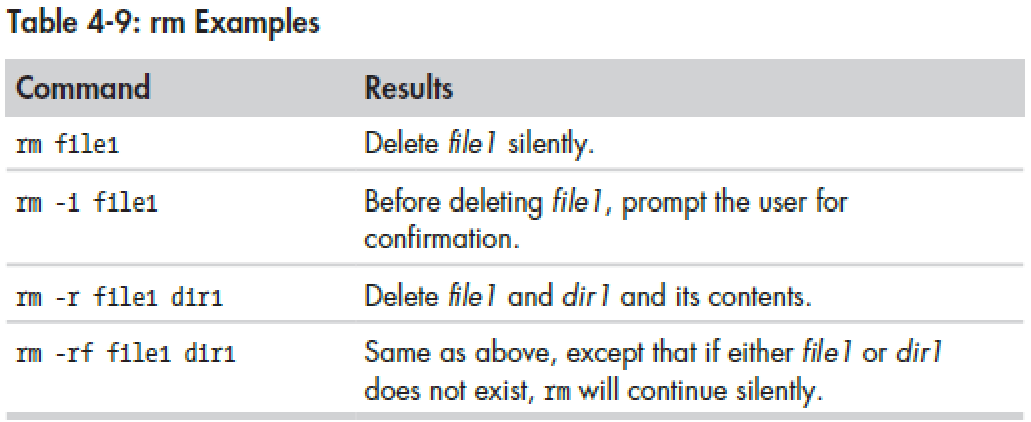


Table 4-9: rm Examples

Command	Results
`rm file1`	Delete *file1* silently.
`rm -i file1`	Before deleting *file1*, prompt the user for confirmation.
`rm -r file1 dir1`	Delete *file1* and *dir1* and its contents.
`rm -rf file1 dir1`	Same as above, except that if either *file1* or *dir1* does not exist, `rm` will continue silently.

ln—Create Links

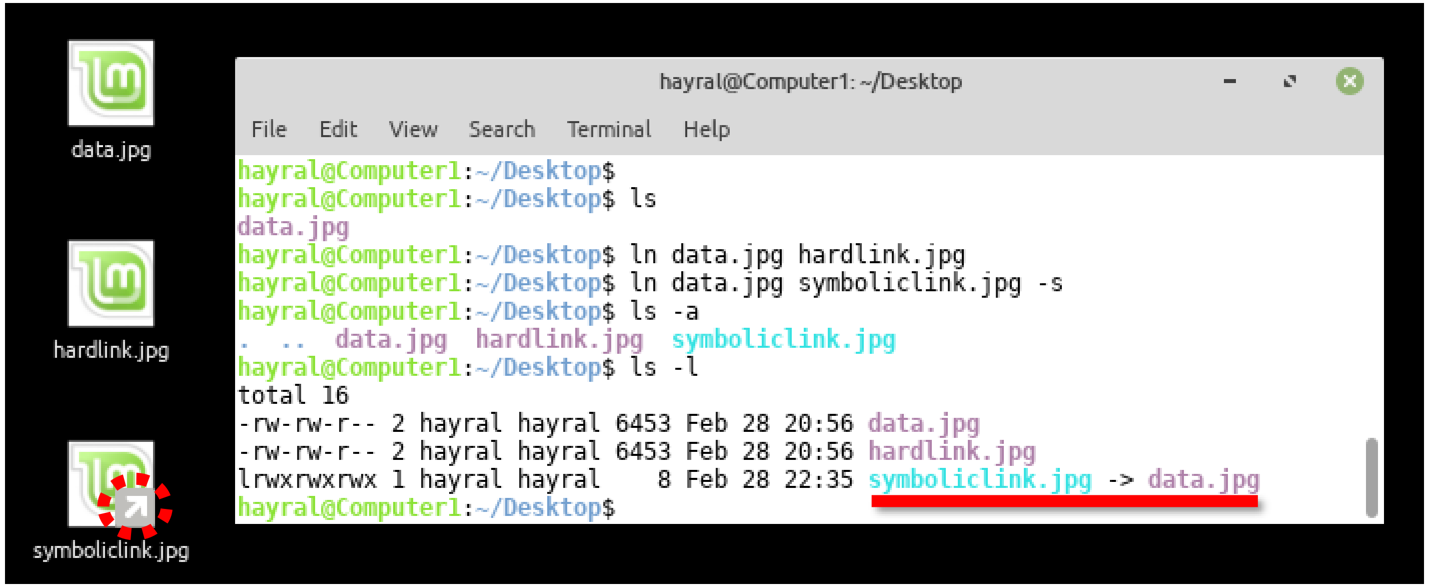
- The ***ln*** command is used to create either hard or symbolic links. It is used in one of two ways:

```
ln file link
```

- to create a **hard link** and

```
ln -s item link
```

- to create a **symbolic link** where *item* is either a file or a directory.


data.jpg
hardlink.jpg
symboliclink.jpg
hayral@Computer1: ~/Desktop
File Edit View Search Terminal Help

```
hayral@Computer1:~/Desktop$
hayral@Computer1:~/Desktop$ ls
data.jpg
hayral@Computer1:~/Desktop$ ln data.jpg hardlink.jpg
hayral@Computer1:~/Desktop$ ln data.jpg symboliclink.jpg -s
hayral@Computer1:~/Desktop$ ls -a
.  ..  data.jpg  hardlink.jpg  symboliclink.jpg
hayral@Computer1:~/Desktop$ ls -l
total 16
-rw-rw-r-- 2 hayral hayral 6453 Feb 28 20:56 data.jpg
-rw-rw-r-- 2 hayral hayral 6453 Feb 28 20:56 hardlink.jpg
lrwxrwxrwx 1 hayral hayral    8 Feb 28 22:35 symboliclink.jpg -> data.jpg
hayral@Computer1:~/Desktop$
```

Hard Links

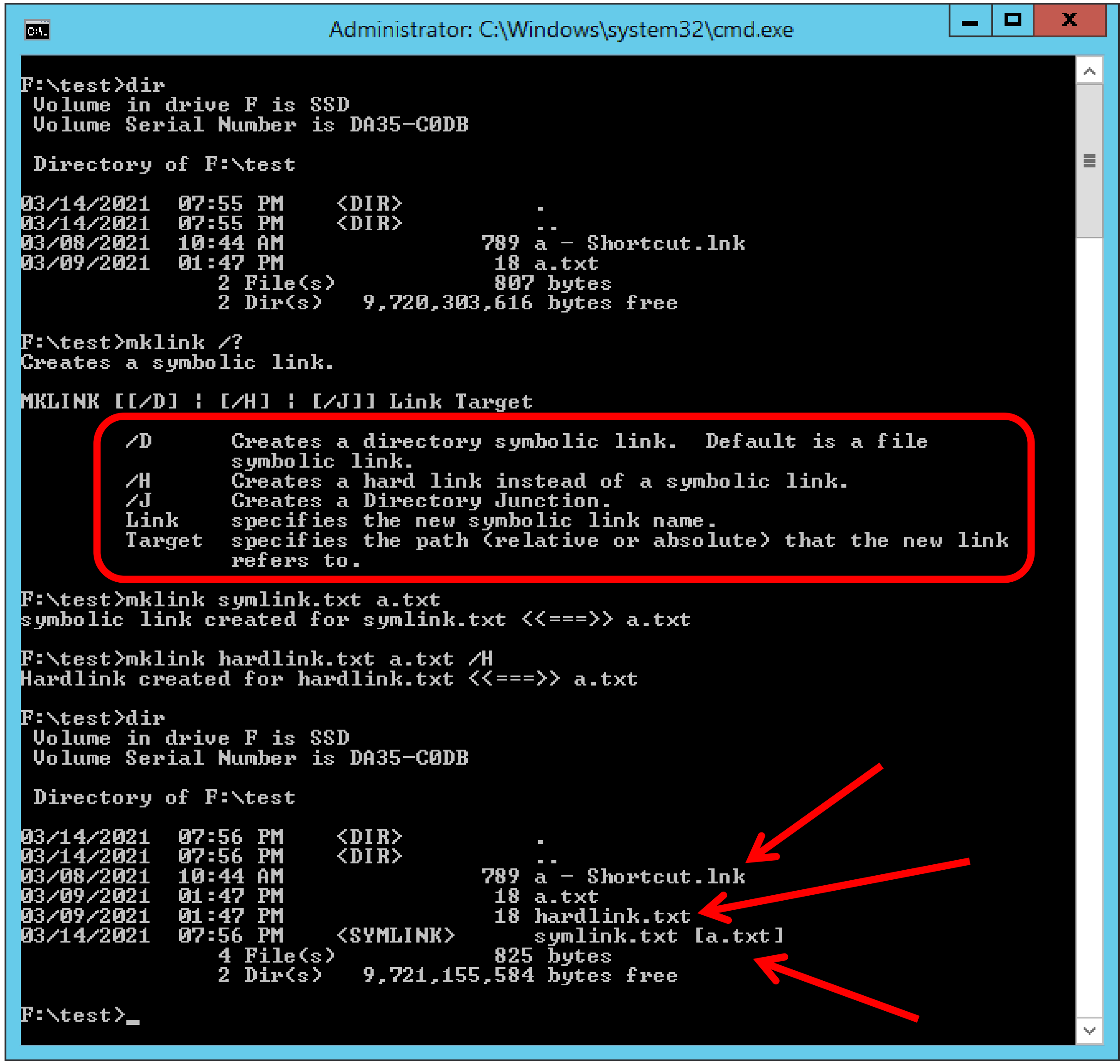
- Hard links are the original Unix way of creating links; symbolic links are more modern.
- By default, every file has a single hard link when created and it gives the file its name.
- When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:
    - A hard link cannot reference a file outside its own filesystem.
        - This means a link cannot reference a file that is not on the same disk partition as the link itself.
    - A hard link cannot reference a directory.
- A hard link is indistinguishable from the file itself. Unlike a directory list containing a symbolic link, a directory list containing a hard link shows no special indication of the link.
- When a hard link is deleted, the link is removed, but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.
- It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links.

Symbolic Links

- Symbolic links were created to overcome the limitations of hard links.
- Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard they operate in much the same way as a Windows shortcut.
- A file pointed to by a symbolic link and the symbolic link itself are largely indistinguishable from one another.
  - For example, if you write something to the symbolic link, the referenced file is written to.
  - However, when you delete a symbolic link, only the link is deleted, not the file itself.
- If the file is deleted before the symbolic link, the link will continue to exist but will point to nothing.
  - In this case, the link is said to be *broken*.
- In many implementations, the ls command will display broken links in a distinguishing color, such as red, to reveal their presence.

Links in Windows OS

- In windows we have five kinds of link:
  - Shortcut
    - Somehow similar to linux sym link
    - It is a binary file, and it is not treated specially on file system level (unlike what linux does)
    - Only Windows GUI interprets it as a link, on console it is just another file
    - Can only be create through GUI
  - Symbolic link
    - Similar to linux sym link: it shows up as a link at file system level
    - Can be on a different partition
  - Hard link
    - Most similar to linux hard link
  - Junction Point
    - No counterpart in linux
  - Directory symbolic link
    - No counterpart in linux



```
F:\test>dir
 Volume in drive F is SSD
 Volume Serial Number is DA35-C0DB

 Directory of F:\test

03/14/2021  07:55 PM    <DIR>          .
03/14/2021  07:55 PM    <DIR>          ..
03/08/2021  10:44 AM               789 a - Shortcut.lnk
03/09/2021  01:47 PM                18 a.txt
               2 File(s)            807 bytes
               2 Dir(s)   9,720,303,616 bytes free

F:\test>mklink /?
Creates a symbolic link.

MKLINK [[/D] | [/H] | [/J]] Link Target

        /D      Creates a directory symbolic link.  Default is a file
                symbolic link.
        /H      Creates a hard link instead of a symbolic link.
        /J      Creates a Directory Junction.
        Link    specifies the new symbolic link name.
        Target  specifies the path (relative or absolute) that the new link
                refers to.

F:\test>mklink symlink.txt a.txt
symbolic link created for symlink.txt <<===>> a.txt

F:\test>mklink hardlink.txt a.txt /H
Hardlink created for hardlink.txt <<===>> a.txt

F:\test>dir
 Volume in drive F is SSD
 Volume Serial Number is DA35-C0DB

 Directory of F:\test

03/14/2021  07:56 PM    <DIR>          .
03/14/2021  07:56 PM    <DIR>          ..
03/08/2021  10:44 AM               789 a - Shortcut.lnk
03/09/2021  01:47 PM                18 a.txt
03/09/2021  01:47 PM                18 hardlink.txt
03/14/2021  07:56 PM    <SYMLINK>      symlink.txt [a.txt]
               4 File(s)            825 bytes
               2 Dir(s)   9,721,155,584 bytes free

F:\test>_
```

Working with Commands

- type—Indicate how a command name is interpreted.
- which—Display which executable program will be executed.
- man—Display a command’s manual page.
- apropos—Display a list of appropriate commands.
- info—Display a command’s info entry.
- whatis—Display a very brief description of a command.
- alias—Create an alias for a command.

What is a command?

A command can be one of four things:

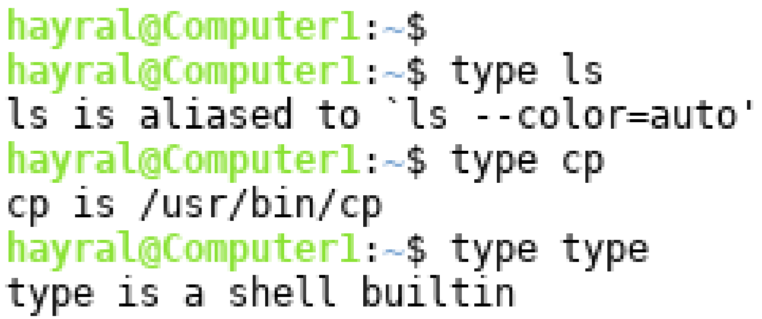
- **An executable program**
  - like all the files in */usr/bin*. ,
  - Programs in this category are
    - *compiled binaries,* such as programs written in C and C++, or
    - programs written in *scripting languages,* such as the shell, Perl, Python, Ruby, and so on.
- **A command built into the shell itself**
  - bash supports a number of commands internally called *shell builtins*.
  - The ***cd*** command, for example, is a shell builtin.
- **A shell function**
  - *Shell functions* are miniature shell scripts incorporated into the *environment*.
- **An alias**
  - An *alias* is a command that we can define ourselves, built from other commands.

type—Display a Command’s Type

- ***type*** command is a shell builtin that displays the kind of command the shell will execute, given a particular command name.
- It works as follows:

```
type command
```

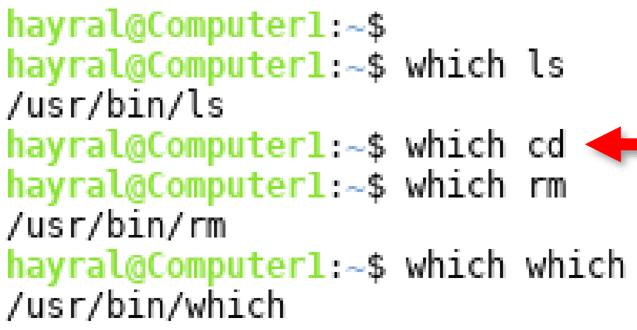
- where *command* is the name of the command you want to examine.



```
hayral@Computer1:~$
hayral@Computer1:~$ type ls
ls is aliased to `ls --color=auto'
hayral@Computer1:~$ type cp
cp is /usr/bin/cp
hayral@Computer1:~$ type type
type is a shell builtin
```

which—Display an Executable’s Location

- Sometimes more than one version of an executable program is installed on a system.
- To determine the exact location of a given executable, ***which*** command is used.
- ***which*** works only for executable programs, not builtins or aliases that are substitutes for actual executable programs.
- When we try to use which on a shell builtin (for example, cd), we get either no response or an error message.



```
hayral@Computer1:~$
hayral@Computer1:~$ which ls
/usr/bin/ls
hayral@Computer1:~$ which cd
hayral@Computer1:~$ which rm
/usr/bin/rm
hayral@Computer1:~$ which which
/usr/bin/which
```

man—Display a Program’s Manual Page

- Most executable programs intended for command-line use provide a formal piece of documentation called a *manual* or *man page*.
- A special paging program called ***man*** is used to view them, like this:

```
man program
```

where *program* is the name of the command to view.

- Man pages vary somewhat in format but generally contain a title, a synopsis of the command’s syntax, a description of the command’s purpose, and a listing and description of each of the command’s options.
- Man pages, however, do not usually include examples, and they are intended as a reference, not a tutorial.

man—Display a Program’s Manual Page

- On most Linux systems, man uses less to display the manual page, so all of the familiar less commands work while displaying the page.
- The “manual” that man displays is broken into sections and covers not only user commands but also system administration commands, programming interfaces, file formats, and more.
- Sometimes we need to look in a specific section of the manual to find what we are looking for.
- This is particularly true if we are looking for a file format that is also the name of a command. compare `man passwd` vs `man 5 passwd`
- If we don’t specify a section number, we will always get the first instance of a match, probably in section 1.
- To specify a section number, we use man like this:

```
man section search_term
```

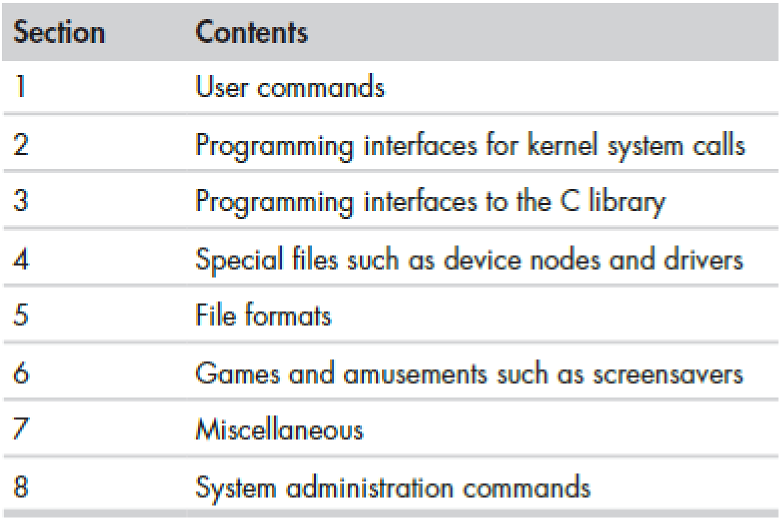


Table 5-1: Man Page Organization

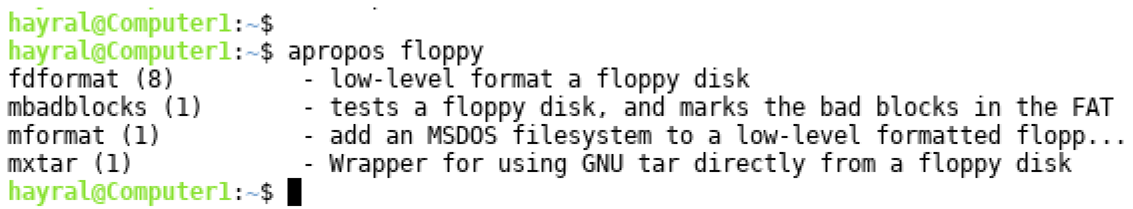
Section	Contents
1	User commands
2	Programming interfaces for kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screensavers
7	Miscellaneous
8	System administration commands

man pages

- man pages are typically installed with the system. Program-specific man pages are added when you install new software packages.
- Be aware of the section definitions when a topic with the same name appears in multiple sections. For example, **passwd** is both a command and a configuration file, so it has entries in both section 1 and section 4 or 5.
- The form **man** *section title* gets you a man page from a particular section. Thus, on most systems, **man sync** gets you the man page for the **sync** command, and **man 2 sync** gets you the man page for the **sync** system call.
- **`man -k`** *`keyword`* or **`apropos`** *`keyword`* prints a list of man pages that have *keyword* in their one-line synopses.

apropos—Display Appropriate Commands

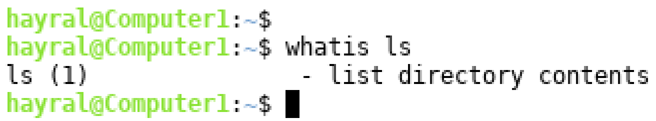
- It is also possible to search the list of man pages for possible matches based on a search term.



```
hayral@Computer1:~$
hayral@Computer1:~$ apropos floppy
fdformat (8)         - low-level format a floppy disk
mbadblocks (1)       - tests a floppy disk, and marks the bad blocks in the FAT
mformat (1)          - add an MSDOS filesystem to a low-level formatted flopp...
mxtar (1)            - Wrapper for using GNU tar directly from a floppy disk
hayral@Computer1:~$ █
```

whatis—Display Description of a Command

- The whatis program displays the name and a one-line description of a man page matching a specified keyword.



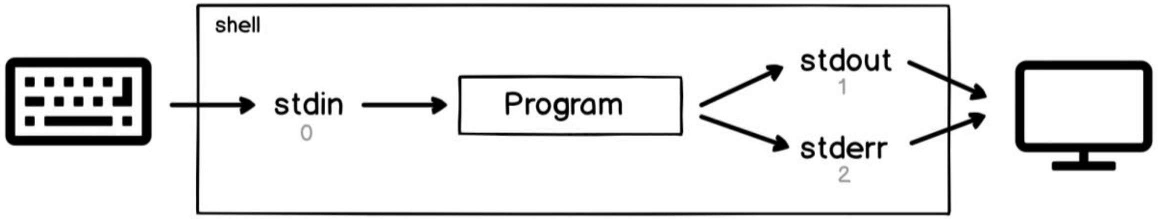
```
hayral@Computer1:~$
hayral@Computer1:~$ whatis ls
ls (1)               - list directory contents
hayral@Computer1:~$ █
```

I/O Redirection

- cat—Concatenate files.
- sort—Sort lines of text.
- uniq—Report or omit repeated lines.
- wc—Print newline, word, and byte counts for each file.
- grep—Print lines matching a pattern.
- 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.

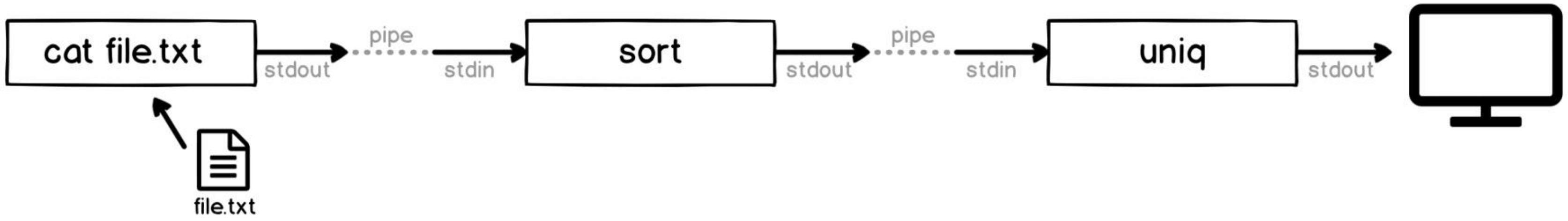
Standard Input, Output, and Error

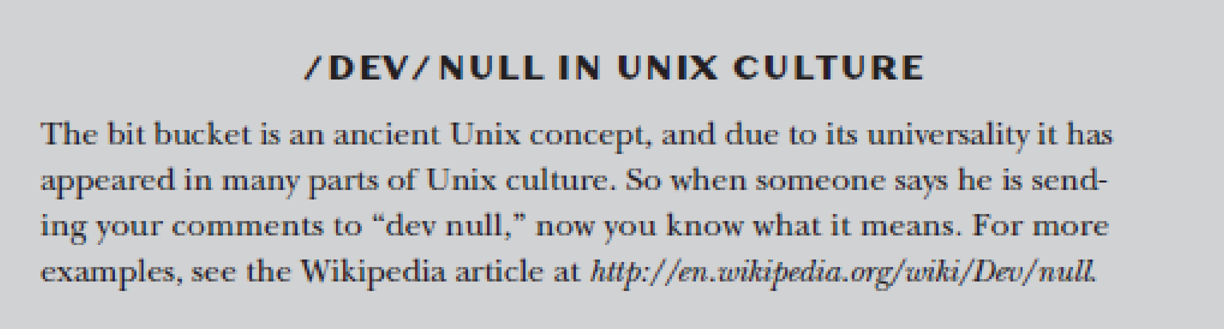
- Many of the programs that we have used so far produce output of some kind.
- This output often consists of two types.
  - First, we have the program's results; that is, the data the program is designed to produce.
  - Second, we have status and error messages that tell us how the program is getting along.
- If we look at a command like ***ls***, we can see that it displays its results and its error messages on the screen.
- Keeping with the Unix theme of "everything is a file," programs such as ls actually send their results to a special file called ***standard output*** (often expressed as ***stdout***) and their status messages to another file called ***standard error*** (***stderr***).
- By default, both standard output and standard error are linked to the screen and not saved into a disk file.
- In addition, many programs take input from a facility called ***standard input*** (***stdin***), which is, by default, attached to the keyboard.
- I/O redirection allows us to change where output goes and where input comes from.
- Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection we can change that.



Redirecting stdin/stdout

- I/O redirection allows us to redefine where standard output goes.
- To redirect standard output to another file instead of the screen, we use the > redirection operator followed by the name of the file.
- Why would we want to do this? → It's often useful to store the output of a command in a file.





/DEV/NULL IN UNIX CULTURE

The bit bucket is an ancient Unix concept, and due to its universality it has appeared in many parts of Unix culture. So when someone says he is sending your comments to "dev null," now you know what it means. For more examples, see the Wikipedia article at *http://en.wikipedia.org/wiki/Dev/null.*

Image Source: https://effective-shell.com/docs/part-2-core-skills/thinking-in-pipelines/

Redirecting Standard Input
cat—Concatenate Files

- Up to now, we haven’t encountered any commands that make use of standard input, so we need to introduce one.

cat—Concatenate Files

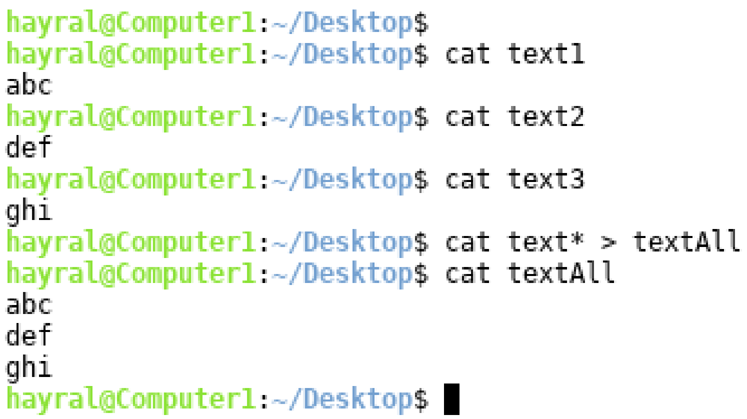
- The cat command reads one or more files and copies them to standard output like so:

```
cat [file...]
```

- Use cases:
  - You can use it to display files without paging.
    - ***cat*** is often used to display short text files.
  - You can use ***cat*** to create short text files from console.
    - `cat > filename`
    - Use CTRL+D as the end-of-file symbol and save the file.
  - Since ***cat*** can accept more than one file as an argument, it can also be used to join files together.
    - `cat file1 file2 file3 > mergedFileName`
    - `cat file* > mergedFileName`

selects matching files in alphabetic order

see also ***tac***, ***rev***



```
hayral@Computer1:~/Desktop$
hayral@Computer1:~/Desktop$ cat text1
abc
hayral@Computer1:~/Desktop$ cat text2
def
hayral@Computer1:~/Desktop$ cat text3
ghi
hayral@Computer1:~/Desktop$ cat text* > textAll
hayral@Computer1:~/Desktop$ cat textAll
abc
def
ghi
hayral@Computer1:~/Desktop$ 
```

Pipelines

- The ability of commands to read data from standard input and send 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.

```
command1 | command2
```

- To fully demonstrate this, we are going to need some commands.
- We can use ***less*** command to display, page by page, the output of any command that sends its results to standard output; for example:

```
ls -l /usr/bin | less
```

- This is very handy; using this technique, we can conveniently examine the output of any command that produces standard output.

Filters

- Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline.
- Frequently, the commands used this way are referred to as ***filters***.
- Filters take input, change it somehow, and then output it.
- The first one we will try is ***sort***.
- Imagine we want to make a combined list of all of the executable programs in */bin* and */usr/bin*, put them in sorted order, and then view the list:

```
ls /bin /usr/bin | sort | less
```

- Since we specified two directories (*/bin* and */usr/bin*), the output of ls would have consisted of two sorted lists, one for each directory.
- By including ***sort*** in our pipeline, we changed the data to produce a single, sorted list.

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 (see the uniq man page for details) and, by default, removes any duplicates from the list.
- So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the */bin* and */usr/bin* directories) we will add ***uniq*** to our pipeline:

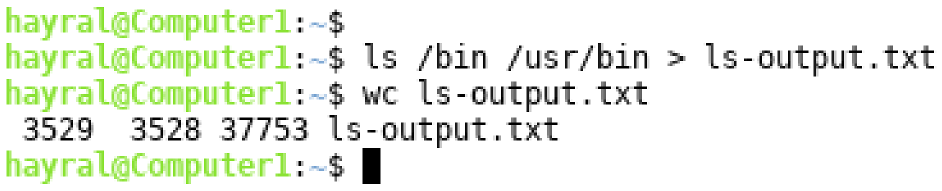
```
ls /bin /usr/bin | sort | uniq | less
```

- In this example, we use ***uniq*** to remove any duplicates from the output of the sort command, if we want to see the list of duplicates instead, we add the -d option to ***uniq*** like so:

```
ls /bin /usr/bin | sort | uniq -d | less
```

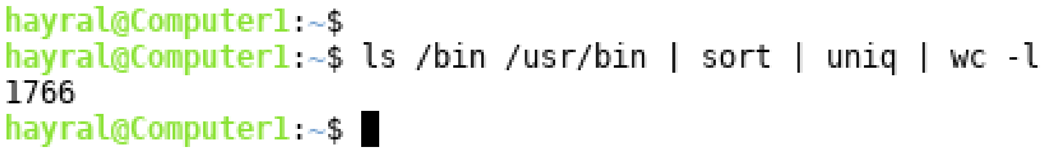
wc—Print Line, Word, and Byte Counts

- The ***wc*** (word count) command is used to display the number of lines, words, and bytes contained in files.



```
hayral@Computer1:~$
hayral@Computer1:~$ ls /bin /usr/bin > ls-output.txt
hayral@Computer1:~$ wc ls-output.txt
 3529  3528 37753 ls-output.txt
hayral@Computer1:~$ 
```

- In this case it prints out three numbers: lines, words, and bytes contained in *ls-output.txt*.
- Like our previous commands, if executed without command-line arguments, wc accepts standard input.
- The -l option limits its output to only report lines.
- Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:



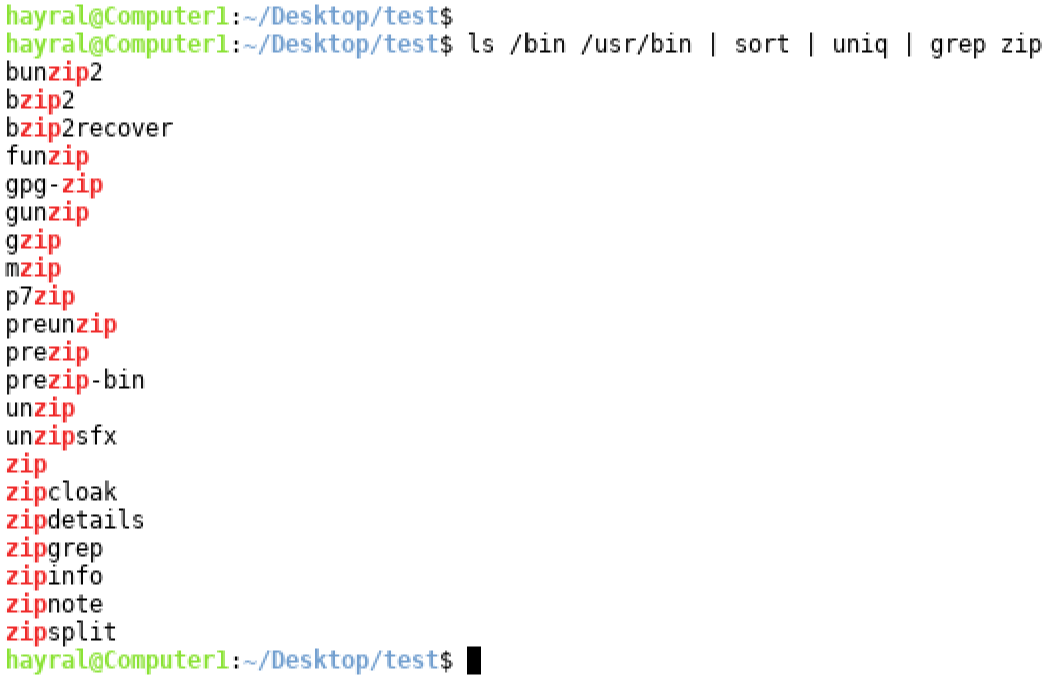
```
hayral@Computer1:~$
hayral@Computer1:~$ ls /bin /usr/bin | sort | uniq | wc -l
1766
hayral@Computer1:~$ 
```

grep—Print Lines Matching a Pattern

- grep is a powerful program used to find text patterns within files, like this:

```
grep pattern [file...]
```

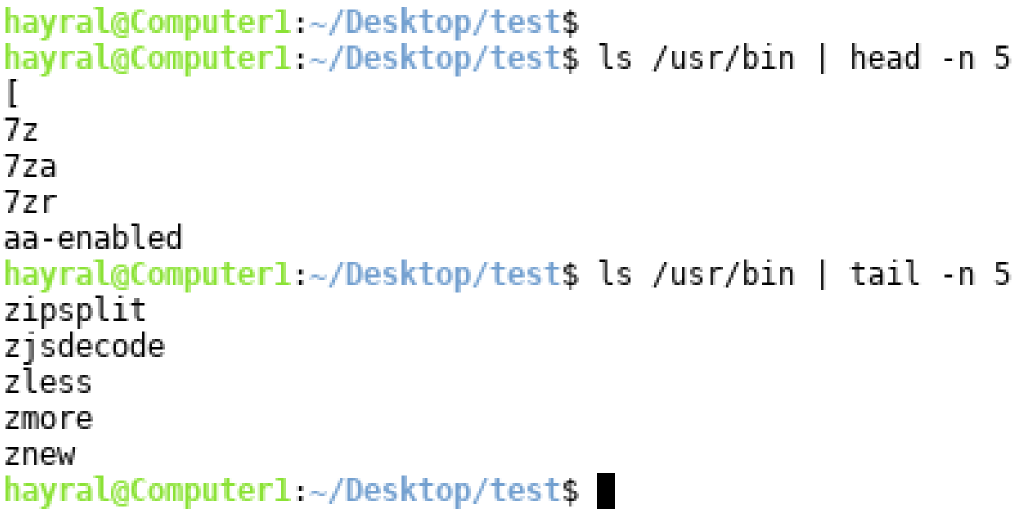
- When grep encounters a “pattern” in the file, it prints out the lines containing it.
- The patterns that grep can match can be very complex, but for now we will concentrate on simple text matches.
- We’ll cover the advanced patterns, called *regular expressions*, later on the course.
- Let’s say we want to find all the files in our list of programs that have the word *zip* in the name.
  - Such a search might give us an idea of which programs on our system have something to do with file compression.
- We would do this:



```
hayral@Computer1:~/Desktop/test$
hayral@Computer1:~/Desktop/test$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
bzip2recover
funzip
gpg-zip
gunzip
gzip
mzip
p7zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
zip
zipcloak
zipdetails
zipgrep
zipinfo
zipnote
zipsplit
hayral@Computer1:~/Desktop/test$
```

head/tail—Print First/Last Part of Files

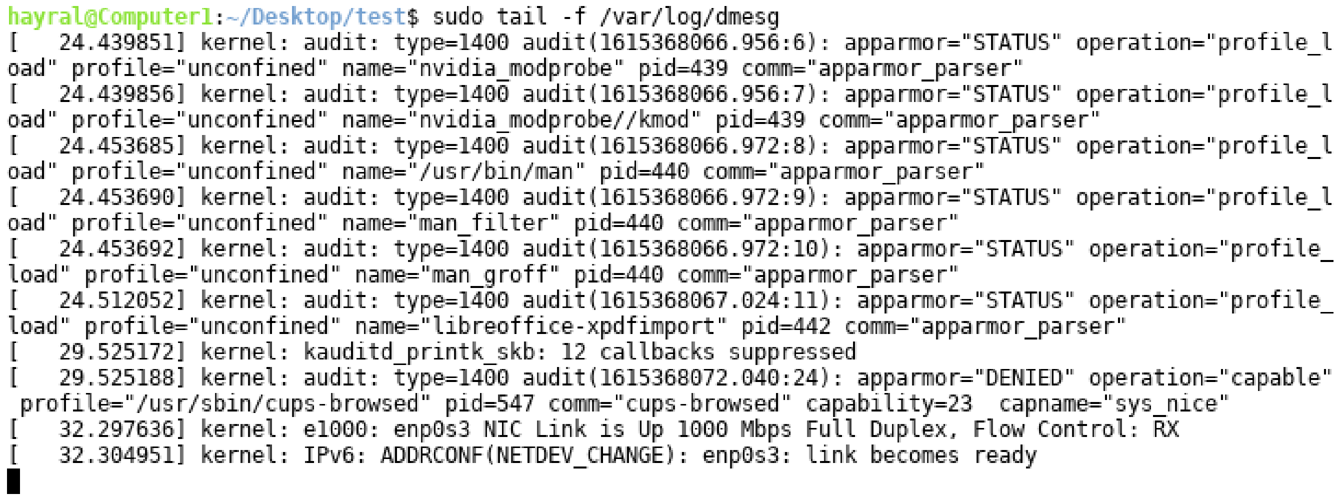
- Sometimes you don’t want all the output from a command. You may want only the first few lines or the last few lines.
- The head command prints the first 10 lines of a file, and the tail command prints the last 10 lines.
- By default, both commands print 10 lines of text, but this can be adjusted with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



```
hayral@Computer1:~/Desktop/test$
hayral@Computer1:~/Desktop/test$ ls /usr/bin | head -n 5
[
7z
7za
7zr
aa-enabled
hayral@Computer1:~/Desktop/test$ ls /usr/bin | tail -n 5
zipsplit
zjsdecode
zless
zmore
znew
hayral@Computer1:~/Desktop/test$ 
```

head/tail—Print First/Last Part of Files

- ***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.
- In the following example, we will look at the *messages* file in */var/log*.
- Superuser privileges are required to do this on some Linux distributions, because the */var/log/messages* file may contain security information.

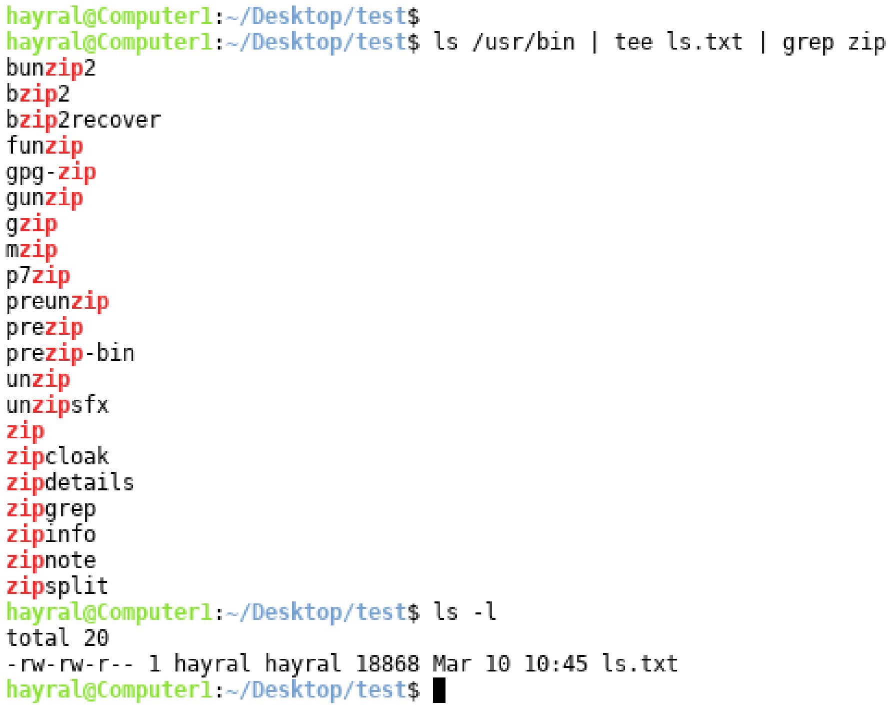


```
hayral@Computer1:~/Desktop/test$ sudo tail -f /var/log/dmesg
[   24.439851] kernel: audit: type=1400 audit(1615368066.956:6): apparmor="STATUS" operation="profile_l
oad" profile="unconfined" name="nvidia_modprobe" pid=439 comm="apparmor_parser"
[   24.439856] kernel: audit: type=1400 audit(1615368066.956:7): apparmor="STATUS" operation="profile_l
oad" profile="unconfined" name="nvidia_modprobe//kmod" pid=439 comm="apparmor_parser"
[   24.453685] kernel: audit: type=1400 audit(1615368066.972:8): apparmor="STATUS" operation="profile_l
oad" profile="unconfined" name="/usr/bin/man" pid=440 comm="apparmor_parser"
[   24.453690] kernel: audit: type=1400 audit(1615368066.972:9): apparmor="STATUS" operation="profile_l
oad" profile="unconfined" name="man_filter" pid=440 comm="apparmor_parser"
[   24.453692] kernel: audit: type=1400 audit(1615368066.972:10): apparmor="STATUS" operation="profile_
load" profile="unconfined" name="man_groff" pid=440 comm="apparmor_parser"
[   24.512052] kernel: audit: type=1400 audit(1615368067.024:11): apparmor="STATUS" operation="profile_
load" profile="unconfined" name="libreoffice-xpdfimport" pid=442 comm="apparmor_parser"
[   29.525172] kernel: kauditd_printk_skb: 12 callbacks suppressed
[   29.525188] kernel: audit: type=1400 audit(1615368072.040:24): apparmor="DENIED" operation="capable"
 profile="/usr/sbin/cups-browsed" pid=547 comm="cups-browsed" capability=23  capname="sys_nice"
[   32.297636] kernel: e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[   32.304951] kernel: IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
█
```

- Using the -f option, tail continues to monitor the file and when new lines are appended, they immediately appear on the display.
- This continues until you type CTRL-C.

tee—Read from Stdin and Output to Stdout and Files

- Linux provides a command called tee which creates a “T” fitting on our pipe.
- The tee program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files.
- This is useful for capturing a pipeline’s contents at an intermediate stage of processing.
- Here we repeat one of our earlier examples, this time including tee to capture the entire directory listing to the file *ls.txt* before grep filters the pipeline’s contents:



```
hayral@Computer1:~/Desktop/test$
hayral@Computer1:~/Desktop/test$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
bzip2recover
funzip
gpg-zip
gunzip
gzip
mzip
p7zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
zip
zipcloak
zipdetails
zipgrep
zipinfo
zipnote
zipsplit
hayral@Computer1:~/Desktop/test$ ls -l
total 20
-rw-rw-r-- 1 hayral hayral 18868 Mar 10 10:45 ls.txt
hayral@Computer1:~/Desktop/test$
```