Basic commands

There are many basic command line utilities that are used to list, examine, and manipulate files and directories. Each has many options. Here is a list of very commonly used ones, that should be in any user's toolbox:

Name	Purpose
ls	List files
cat	Type out (concatenate) the files
rm	Remove files
mv	Rename (move) files
mkdir	Create directories
rmdir	Remove directories
file	Show file types
ln	Create symbolic and hard links
tail	Look at the tail end of the file
head	Look at the beginning of the file
less	Look at the file, one screenful at a time
more	Look at the file, one screenful at a time
touch	Either create an empty file, or update the file modification time
wc	Count lines, words, and bytes in a file

Finding Filles: find and locate commands

The find command line utility provides an extremely powerful and flexible method for locating files based on their properties, including name. It does not search the interior of files for patterns, etc.; that is more the province of **grep** and its variations.

The general form of a **find** command is:

```
1  $ find [location] [criteria] [actions]
```

where there are three classes of arguments, each of which may be omitted. If no location is given, the current directory **(.)** is assumed; if no criteria are given, all files are displayed; and, if no actions are given, only a listing of the names is given.

There are many logical expressions which can be used for criteria. For example, the command:

```
1  $ find /etc -name "*.conf"
```

will print out the names of all files in the **/etc** directory and its descendants, recursively, that end in **.conf**. To specify a simple action request:

```
1  $ find /etc -name "*.conf" -ls
```

will print out a long listing, not just the names.

A little more complicated example is the following:

```
$ find /tmp /etc -name "*.conf" -or -newer /tmp/.X0-lock -ls
```

will look in subdirectories under **/etc** and **/tmp** for files whose names end in **.conf**, or are newer than **/tmp/.X0-lock** and print out a long listing.

You can perform actions with the **-exec** option, as in:

```
$ find . -name "*~" -exec rm {} ';'
```

where **{}** is a fill in for the files to be operated on, and **';'** indicates the end of the command. This can be unwieldy and one often pipes into the **xargs** program, as in:

```
$ find . -name "*~" | xargs rm
```

which accomplishes the same action. A third way to do the same action would be:

```
$ for names in $(find . -name "*~" ) ; do rm $names ; done
```

If a filename has a blank space in it (or some other special characters), some of the previous commands will fail.

If a filename has a blank space in it (or some other special characters), some of the previous commands will fail.

It is generally a disfavored practice to utilize such file names in UNIX-like operating systems, but it is not uncommon for such files to exist, either in files brought in from other systems, or from applications which are also used in other systems.

In such a case, the following variant will work just fine:

```
$ find . -name "*~" -print0 | xargs -0 rm
```

as will the command that uses **-exec rm {} ';'**.

There are many options to **find**, especially regarding selection of files to display. This can be done based on size, time of creation or access, type of file, owner, etc. A quick synopsis is provided by **find --help**:

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>find --help
Usage: find [-H] [-L] [-P] [-Olevel] [-D help|tree|search|stat|rates|opt|exec] [path...] [expression]

default path is the current directory; default expression is -print
expression may consist of: operators, options, tests, and actions:

operators (decreasing precedence; -and is implicit where no others are given):
      ( EXPR )   ! EXPR   -not EXPR   EXPR1 -a EXPR2   EXPR1 -and EXPR2
      EXPR1 -o EXPR2   EXPR1 -or EXPR2   EXPR1 , EXPR2

positional options (always true): -daystart -follow -regextype

normal options (always true, specified before other expressions):
      -depth --help -maxdepth LEVELS -mindepth LEVELS -mount -noleaf
      --version -xautofs -xdev -ignore_readdir_race -noignore_readdir_race

tests (N can be +N or -N or N): -amin N -anewer FILE -atime N -cmin N
      -cnewer FILE -ctime N -empty -false -fstype TYPE -gid N -group NAME
      -ilname PATTERN -iname PATTERN -inum N -iwholename PATTERN -iregex PATTERN
      -links N -lname PATTERN -mmin N -mtime N -name PATTERN -newer FILE
      -nouser -nogroup -path PATTERN -perm [-/]MODE -regex PATTERN
      -readable -writable -executable
      -wholename PATTERN -size N[bcwkMG] -true -type [bcdpflsD] -uid N
      -used N -user NAME -xtype [bcdpfls]
      -context CONTEXT


actions: -delete -print0 -printf FORMAT -fprintf FILE FORMAT -print
      -fprint0 FILE -fprint FILE -ls -fls FILE -prune -quit
      -exec COMMAND ; -exec COMMAND {} + -ok COMMAND ;
      -execdir COMMAND ; -execdir COMMAND {} + -okdir COMMAND ;

Report (and track progress on fixing) bugs via the findutils bug-reporting
page at http://savannah.gnu.org/ or, if you have no web access, by sending
email to <bug-findutils@gnu.org>.
c7:/tmp>
```

Another method of locating files is provided by the **locate** command, which searches your entire filesystem (except for paths which have been excluded) and works off a database that is updated periodically with **updatedb**. Thus, it is very fast.

Thus, the command:

```
1  $ locate .conf
```

will find all files on your system that have **.conf** in them.

A quick synopsis is provided by **locate --help**:

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>locate --help
c7:/tmp>locate --help
Usage: locate [OPTION]... [PATTERN]...
Search for entries in a mlocate database.

  -A, --all                only print entries that match all patterns
  -b, --basename           match only the base name of path names
  -c, --count              only print number of found entries
  -d, --database DBPATH    use DBPATH instead of default database (which is
                           /var/lib/mlocate/mlocate.db)
  -e, --existing           only print entries for currently existing files
  -L, --follow             follow trailing symbolic links when checking file
                           existence (default)
  -h, --help               print this help
  -i, --ignore-case        ignore case distinctions when matching patterns
  -l, --limit, -n LIMIT    limit output (or counting) to LIMIT entries
  -m, --mmap               ignored, for backward compatibility
  -P, --nofollow, -H       don't follow trailing symbolic links when checking file
                           existence
  -0, --null               separate entries with NUL on output
  -S, --statistics         don't search for entries, print statistics about each
                           used database
  -q, --quiet              report no error messages about reading databases
  -r, --regexp REGEXP      search for basic regexp REGEXP instead of patterns
      --regex              patterns are extended regexps
  -s, --stdio              ignored, for backward compatibility
  -V, --version            print version information
  -w, --wholename          match whole path name (default)

Report bugs to mitr@redhat.com.
c7:/tmp>
```

locate will only find files that were already in existence the last time the database was updated. On most systems, this is done by a background **cron** job, usually daily. To force an update, you need to do:

```
$ sudo updatedb
```

grep Command

grep is a workhorse command line utility whose basic job is to search files for patterns and print out matches according to specified options.

Its name stands for global regular expression print, which points out that it can do more than just match simple strings; it can work with more complicated regular expressions which can contain wildcards and other special attributes.

The simplest example of using **grep** would be:

```
$ grep pig file

pig
dirty pig
pig food
```

which finds three instances of the string "pig" in file.

As an example:

```
$ grep -i -e pig -e dog -r .
```

will search all files in the current directory and those below it for the strings "pig" or "dog", ignoring case.

will search all files in the current directory and those below it for the strings "pig" or "dog", ignoring case.

If we try to explore the use of regular expressions in detail, it would be a large topic, but here are some examples:

```
# print all lines that start with "dog"
$ grep "^dog" file

# print all lines that end with "dog"
$ grep "dog$" file

# print all lines that end with "dog"
$ grep d[a-p] file
```

grep has many options; some of the most important are:

Option	Meaning
-i	Ignore case
-v	Invert match
-n	Print line number
-H	Print filename
-a	Treat binary files as text
-I	Ignore binary files
-r	Recurse through subdirectories
-l	Print out names of all files that contain matches
-L	Print out names of all files that do not contain matches
-c	Print out number of matching lines only
-e	Use the following pattern; useful for multiple strings and special characters

sed Command

sed stands for stream editor. Its job is to make substitutions and other modifications in files and in streamed output.

Any of the following methods will change all first instances of the string "pig" with "cow" for each line of **file**, and put the results in **newfile**:

```
1 $ sed s/pig/cow/ file > newfile
2 $ sed s/pig/cow/ < file > newfile
3 $ cat file | sed s/pig/cow/ > newfile
```

where the **s** stands for substitute. If you want to change all instances, you have to add the **g** (global) qualifier, as in:

```
1 $ sed s/pig/cow/g file > newfile
```

The **/** characters are used to delimit the new and old strings. You can choose to use another character, as in:

```
1 $ sed s:pig:cow:g file > newfile
```

Some of the complications come in when you want to use special characters in the strings to be searched for or inserted. For example, suppose you want to replace all back slashes with forward slashes:

```
$ sed s/'\\'/'\/'/g file > newfile
```

It is never a bad idea to put the strings in either single or double quotes, the main difference being that environment variables and escaped special characters in double quotes are expanded out while they are not in single quotes, as in:

```
$ echo "$HOME"
/home/coop

$ echo '$HOME'
$HOME
```

If you want to make multiple simultaneous substitutions, you need to use the **-e** option, as in:

```
$ sed -e s/"pig"/"cow"/g -e s/"dog"/"cat"/g < file > newfile
```

and you can work directly on streams generated from commands, as in:

```
$ echo hello | sed s/"hello"/"goodbye"/g

goodbye
```

If you have a lot of commands, you can put them in a file and apply the **-f** option, as in:

```
$ cat scriptfile

s/pig/cow/g
s/dog/cat/g
s/frog/toad/g

$ sed -f scriptfile < file > newfile
```

Command Line Tools for Text File Manipulation

Manipulating Text Files

- Irrespective of the role you play with Linux (system administrator, developer or user), you often need to browse through and parse text files, and/or extract data from them
- These are **file manipulation operations** and it is essential for the Linux user to become adept at performing certain operations on files
- Most of the time, such file manipulation is done at the command line, which allows users to perform tasks more efficiently than while using a GUI; the command line is also more suitable for automating often executed tasks
- Experienced system administrators write customized scripts to accomplish such repetitive tasks, standardized for each particular environment

Command Line Utilities

The basic Linux command line utilities are:

- **cat**
- **echo**
- **head**
- **tail**
- **sed**
- **awk**

cat Command

cat is short for concatenate and is one of the most frequently used Linux command line utilities. It is often used to read and print files, as well as for simply viewing file contents. To view a file, use the following command:

```
$ cat <filename>
```

For example, **cat readme.txt** will display the contents of **readme.txt** on the terminal. However, the main purpose of **cat** is often to combine (concatenate) multiple files together. You can perform the actions listed in the table using **cat**.

The **tac** command (**cat** spelled backwards) prints the lines of a file in reverse order. Each line remains the same, but the order of lines is inverted. The syntax of **tac** is exactly the same as for **cat**, as in:

```
$ tac file
$ tac file1 file2 > newfile
```

Command	Usage
cat file1 file2	Concatenate multiple files and display the output; i.e. the entire content of the first file is followed by that of the second file
cat file1 file2 > newfile	Combine multiple files and save the output into a new file
cat file >> existingfile	Append a file to the end of an existing file
cat > file	Any subsequent lines typed will go into the file, until ***Ctrl-D*** is typed
cat >> file	Any subsequent lines are appended to the file, until ***Ctrl-D*** is typed

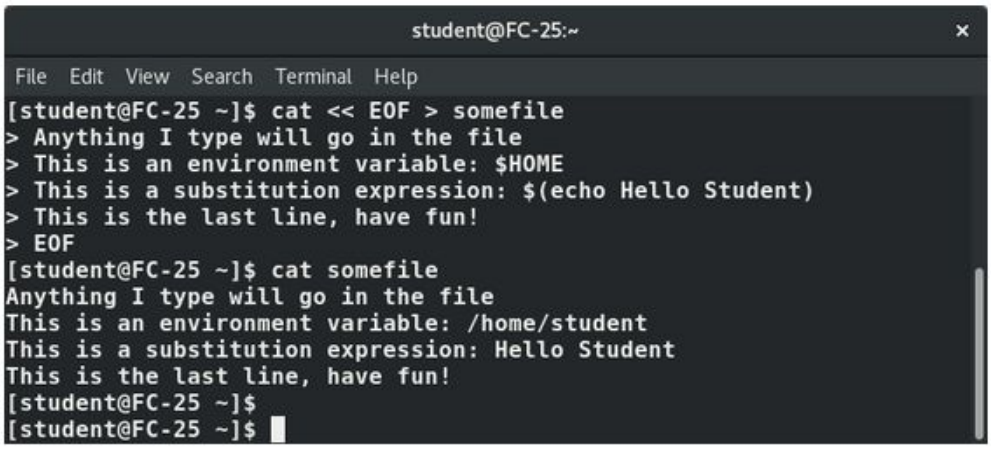
cat can be used to read from standard input (such as the terminal window) if no files are specified. You can use the **>** operator to create and add lines into a new file, and the **>>** operator to append lines (or files) to an existing file. We mentioned this when talking about how to create files without an editor.

To create a new file, at the command prompt, type **cat > <filename>** and press the ***Enter*** key.

This command creates a new file and waits for the user to edit/enter the text. After you finish typing the required text, press ***Ctrl-D*** at the beginning of the next line to save and exit the editing.

Another way to create a file at the terminal is **cat > <filename> << EOF**. A new file is created and you can type the required input. To exit, enter **EOF** at the beginning of a line.

Note that **EOF** is case sensitive. One can also use another word, such as **STOP**.



echo Command

echo displays (echoes) text. It is used simply, as in:

```
$ echo string
```

echo can be used to display a string on standard output (i.e. the terminal) or to place in a new file (using the **>** operator) or append to an already existing file (using the **>>** operator).

The **–e** option, along with the following switches, is used to enable special character sequences, such as the newline character or horizontal tab.

- **\n** represents newline
- **\t** represents horizontal tab

echo is particularly useful for viewing the values of environment variables (built-in shell variables). For example, **echo $USERNAME** will print the name of the user who has logged into the current terminal.

The following table lists **echo** commands and their usage:

Command	Usage
echo string > newfile	The specified string is placed in a new file
echo string >> existingfile	The specified string is appended to the end of an already existing file
echo $variable	The contents of the specified environment variable are displayed

Working on Large Files

`less` and `somefile`

- One can use **`less`** to view the contents of such a large file, scrolling up and down page by page, without the system having to place the entire file in memory before starting; this is much faster than using a text editor
- Viewing **`somefile`** can be done by typing either of the two following commands:

```
$ less somefile
$ cat somefile | less
```

- By default, man pages are sent through the **`less`** command
- You may have encountered the older **`more`** utility which has the same basic function but fewer capabilities: i.e. less is more!

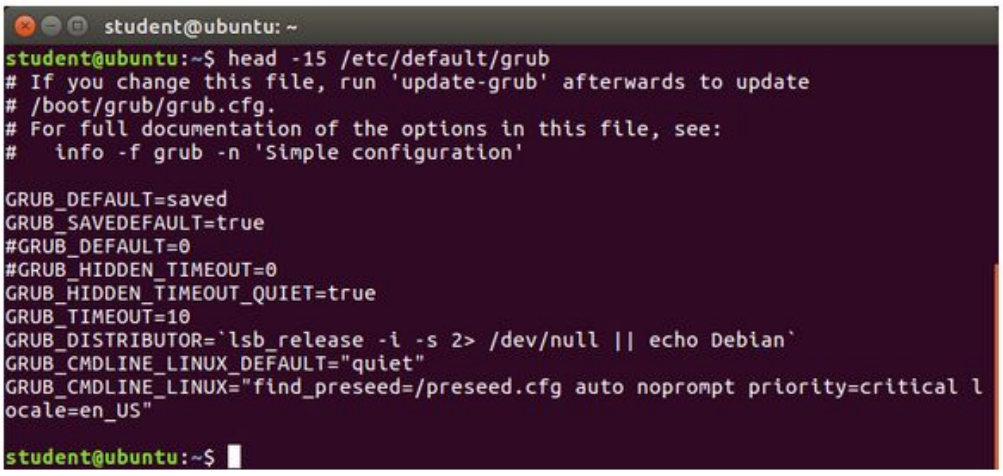
head Command

head reads the first few lines of each named file (10 by default) and displays it on standard output. You can give a different number of lines in an option.

For example, if you want to print the first 5 lines from **grub.cfg**, use the following command:

```
1  $ head -n 5 grub.cfg
```

You can also just say **head -5 grub.cfg**.



tail Command

tail prints the last few lines of each named file and displays it on standard output. By default, it displays the last 10 lines. You can give a different number of lines as an option. **tail** is especially useful when you are troubleshooting any issue using log files, as you probably want to see the most recent lines of output.

For example, to display the last 15 lines of **somefile.log**, use the following command:

```
1 $ tail -n 15 somefile.log
```

You can also just say **tail -15 somefile.log**.

To continually monitor new output in a growing log file:

```
1 $ tail -f somefile.log
```

This command will continuously display any new lines of output in **somefile.log** as soon as they appear. Thus, it enables you to monitor any current activity that is being reported and recorded.

Viewing Compressed Files

When working with compressed files, many standard commands cannot be used directly. For many commonly-used file and text manipulation programs, there is also a version especially designed to work directly with compressed files. These associated utilities have the letter "**z**" prefixed to their name. For example, we have utility programs such as **zcat**, **zless**, **zdiff** and **zgrep**.

Here is a table listing some **z** family commands:

Command	Description
$ zcat compressed-file.txt.gz	To view a compressed file
$ zless somefile.gz or $ zmore somefile.gz	To page through a compressed file
$ zgrep -i less somefile.gz	To search inside a compressed file
$ zdiff file1.txt.gz file2.txt.gz	To compare two compressed files

Note that if you run **zless** on an uncompressed file, it will still work and ignore the decompression stage. There are also equivalent utility programs for other compression methods besides **gzip.**

For example, we have **bzcat** and **bzless** associated with **bzip2**, and **xzcat** and **xzless** associated with **xz**.

Introduction to sed and awk

Introduction to **sed** and **awk**

- It is very common to create and then repeatedly edit and/or extract contents from a file; in order to perform such operations you can use **sed** and **awk**
- Many Linux users and administrators will write scripts using comprehensive scripting languages such as **Python** and **perl**, rather than use **sed** and **awk** (and some other utilities we will discuss later)
- Using such utilities is certainly fine in most circumstances; you should always feel free to use the tools you are experienced with
- However, the utilities that are described here are much lighter; i.e. they use fewer system resources, and execute faster
- There are situations (such as during booting the system) where a lot of time would be wasted using the more complicated tools, and the system may not even be able to run them; the simpler tools will always be needed

sed

- **sed** is a powerful text processing tool and one of the oldest, earliest and most popular UNIX utilities; its name is an abbreviation for stream editor
- It is used to modify the contents of a file, usually placing the contents into a new file, and it can also filter text, as well as perform substitutions in data streams
- How it works:
  - Data from an input source/file (or stream) is taken and moved to a working space
  - The entire list of operations/modifications is applied over the data in the working space and the final contents are moved to the standard output space (or stream)

awk

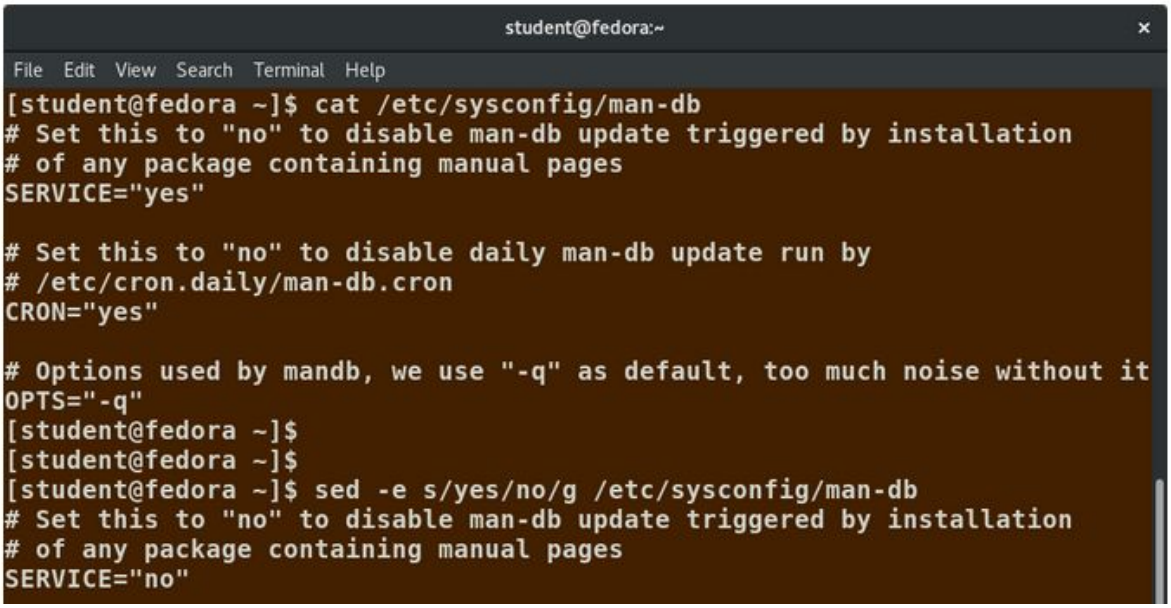
- **awk** was created at Bell Labs in the 1970s and derived its name from the last names of its authors: Alfred Aho, Peter Weinberger, and Brian Kernighan
  - Used to extract and then print specific contents of a file and is often used to construct reports
  - A powerful utility and interpreted programming language
  - Used to manipulate data files, retrieving, and processing text
  - Works well with fields (containing a single piece of data, essentially a column) and records (a collection of fields, essentially a line in a file)

sed Command syntax and basics

You can invoke **sed** using commands like those listed in the accompanying table.

Command	Usage
sed -e command <filename>	Specify editing commands at the command line, operate on file and put the output on standard out (e.g. the terminal)
sed -f scriptfile <filename>	Specify a scriptfile containing sed commands, operate on file and put output on standard out

The **-e** command option allows you to specify multiple editing commands simultaneously at the command line. It is unnecessary if you only have one operation invoked.



Now that you know that you can perform multiple editing and filtering operations with **sed**, let's explain some of them in more detail. The table explains some basic operations, where **pattern** is the current string and **replace_string** is the new string:

Command	Usage
sed s/pattern/replace_string/ file	Substitute first string occurrence in every line
sed s/pattern/replace_string/g file	Substitute all string occurrences in every line
sed 1,3s/pattern/replace_string/g file	Substitute all string occurrences in a range of lines
sed -i s/pattern/replace_string/g file	Save changes for string substitution in the same file

You must use the **-i** option with care, because the action is not reversible. It is always safer to use **sed** without the **–i** option and then replace the file yourself, as shown in the following example:

```
1  $ sed s/pattern/replace_string/g file1 > file2
```

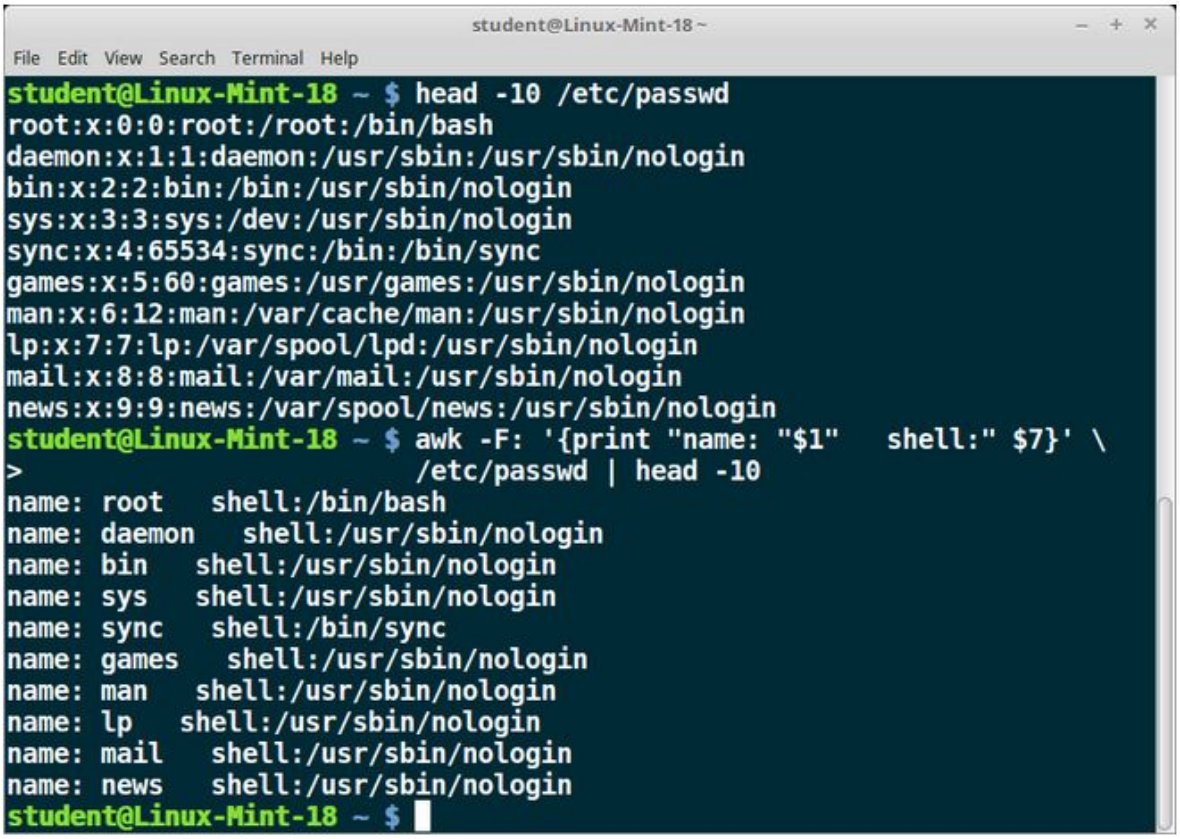
The above command will replace all occurrences of **pattern** with **replace_string** in **file1** and move the contents to **file2**. The contents of **file2** can be viewed with **cat file2**. If you approve, you can then overwrite the original file with **mv file2 file1**.

Example: To convert **01/02/... to JAN/FEB/...**

```
1  sed -e 's/01/JAN/' -e 's/02/FEB/' -e 's/03/MAR/' -e 's/04/APR/' -e 's/05/MAY/' \
2      -e 's/06/JUN/' -e 's/07/JUL/' -e 's/08/AUG/' -e 's/09/SEP/' -e 's/10/OCT/' \
3      -e 's/11/NOV/' -e 's/12/DEC/'
```

awk Command syntax and basics

awk is invoked as shown in the following screenshot:



As with **sed**, short **awk** commands can be specified directly at the command line, but a more complex script can be saved in a file that you can specify using the **-f** option.

Command	<header>
awk 'command' file	Specify a command directly at the command line
awk -f scriptfile file	Specify a file that contains the script to be executed

The table below explains the basic tasks that can be performed using **awk**. The input file is read one line at a time, and, for each line, **awk** matches the given pattern in the given order and performs the requested action. The **-F** option allows you to specify a particular field separator character. For example, the **/etc/passwd** file uses "**:**" to separate the fields, so the **-F:** option is used with the **/etc/passwd** file.

The command/action in **awk** needs to be surrounded with apostrophes (or single-quote (')). **awk** can be used as follows:

Command	Usage
awk '{ print $0 }' /etc/passwd	Print entire file
awk -F: '{ print $1 }' /etc/passwd	Print first field (column) of every line, separated by a space
awk -F: '{ print $1 $7 }' /etc/passwd	Print first and seventh field of every line

File Manipulation Utilities

File Manipulation Utilities

- In managing your files, you may need to perform many tasks, such as sorting data and copying data from one location to another
- Linux provides several file manipulation utilities that you can use while working with text files, including:
  - **sort**
  - **uniq**
  - **paste**
  - **join**
  - **split**

sort Command

sort can be used as follows:

Syntax	Usage
sort <filename>	Sort the lines in the specified file, according to the characters at the beginning of each line
cat file1 file2 | sort	Combine the two files, then sort the lines and display the output on the terminal
sort -r <filename>	Sort the lines in reverse order
sort -k 3 <filename>	Sort the lines by the 3rd field on each line instead of the beginning

When used with the **-u** option, **sort** checks for unique values after sorting the records (lines). It is equivalent to running **uniq** (which we shall discuss) on the output of **sort**.

```
File  Edit  View  Search  Terminal  Help
c7:/etc/default>cat grub
GRUB_TIMEOUT="3"
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT="saved"
GRUB_DISABLE_SUBMENU="true"
GRUB_TERMINAL_OUTPUT="console"
#GRUB_CMDLINE_LINUX="vconsole.keymap=us crashkernel=auto  vconsole.font=latarcyrheb-sun16 rhgb quiet
"
GRUB_CMDLINE_LINUX="rhgb quiet"
GRUB_DISABLE_RECOVERY="true"
GRUB_TERMINAL=gfxterm
GRUB_BACKGROUND="/boot/despair.jpg"
c7:/etc/default>
c7:/etc/default>
c7:/etc/default>sort grub
GRUB_BACKGROUND="/boot/despair.jpg"
GRUB_CMDLINE_LINUX="rhgb quiet"
#GRUB_CMDLINE_LINUX="vconsole.keymap=us crashkernel=auto  vconsole.font=latarcyrheb-sun16 rhgb quiet
"
GRUB_DEFAULT="saved"
GRUB_DISABLE_RECOVERY="true"
GRUB_DISABLE_SUBMENU="true"
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_TERMINAL=gfxterm
GRUB_TERMINAL_OUTPUT="console"
```

uniq Command

uniq removes duplicate consecutive lines in a text file and is useful for simplifying the text display.

Because **uniq** requires that the duplicate entries must be consecutive, one often runs **sort** first and then pipes the output into **uniq**; if **sort** is used with the **-u** option, it can do all this in one step.

To remove duplicate entries from multiple files at once, use the following command:

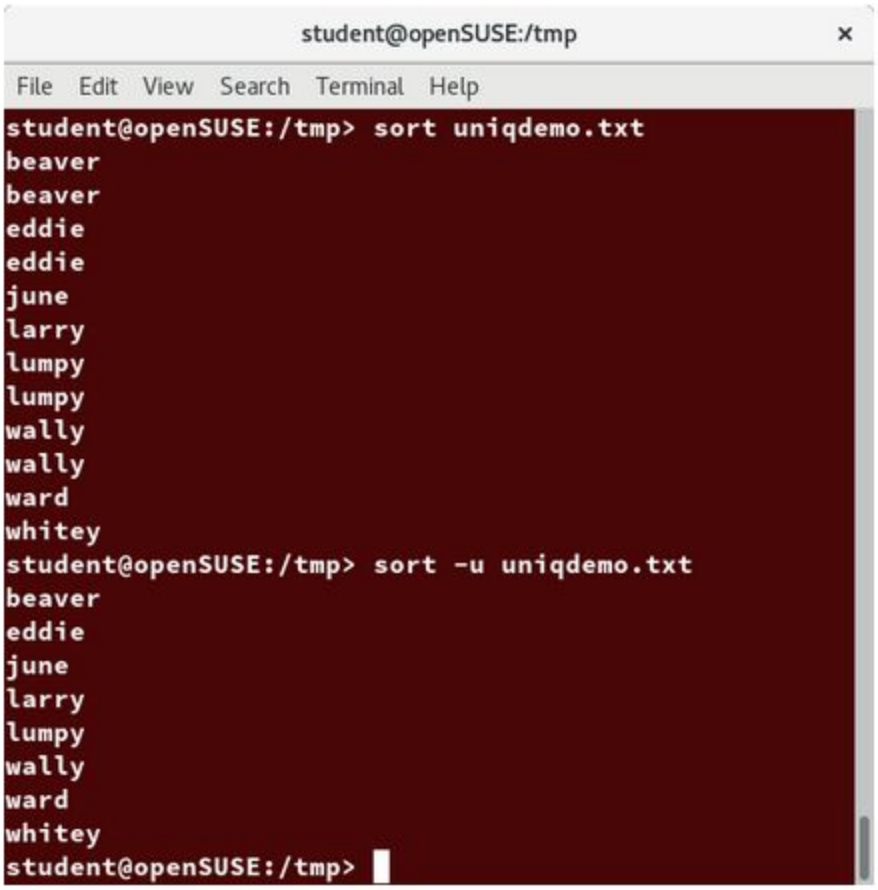
```
1  sort file1 file2 | uniq > file3
```

or

```
1  sort -u file1 file2 > file3
```

To count the number of duplicate entries, use the following command:

```
1  uniq -c filename
```

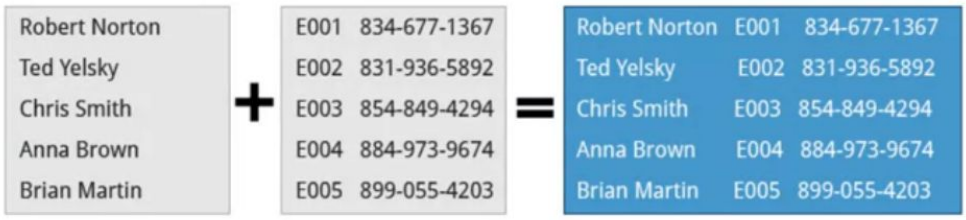


Introduction to paste,join and split Command

Example

Suppose you have a file that contains the full name of all employees and another file that lists their phone numbers and Employee IDs. You want to create a new file that contains all the data listed in three columns: name, employee ID, and phone number.

How can you do this effectively without investing too much time?



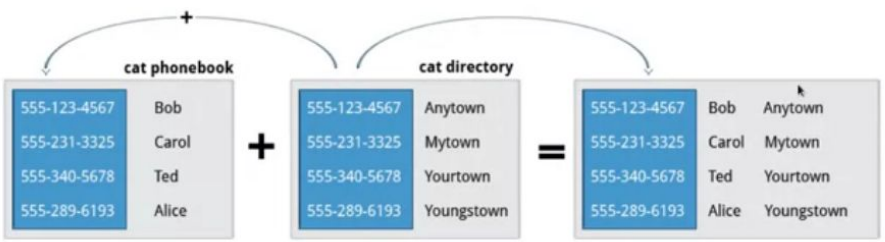
`paste`

- **`paste`** can be used to create a single file containing all three columns
- The different columns are identified based on delimiters (spacing used to separate two fields); for example, delimiters can be a blank space, a tab, or an Enter
- In the image provided, a single space is used as the delimiter in all files
- **`paste`** accepts the following options:
  - **`-d`** delimiters: specify a list of delimiters to be used instead of tabs for separating consecutive values on a single line; each delimiter is used in turn and when the list has been exhausted, **`paste`** begins again at the first delimiter
  - **`-s`** causes **`paste`** to append the data in series rather than in parallel; that is, in a horizontal rather than vertical fashion

Example

Suppose you have two files with some similar columns. You have saved employees' phone numbers in two files, one with their first name and the other with their last name. You want to combine the files without repeating the data of common columns.

How do you achieve this?

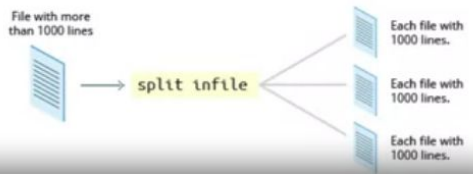


join

- This task can be achieved using **join**, which is essentially an enhanced version of **paste**
- It first checks whether the files share common fields, such as names or phone numbers, and then joins the lines in two files based on a common field

split

- **split** is used to break up (or split) a file into equal-sized segments for easier viewing and manipulation, and is generally used only on relatively large files
- By default, **split** breaks up a file into 1000-line segments; the original file remains unchanged, and a set of new files with the same name plus an added prefix is created
  - By default, the **x** prefix is added
  - To split a file into segments, use the command **split infile**
  - To split a file into segments using a different prefix, use the command **split infile <Prefix>**



paste Command

paste can be used to combine fields (such as name or phone number) from different files, as well as combine lines from multiple files. For example, line one from **file1** can be combined with line one of **file2**, line two from **file1** can be combined with line two of **file2**, and so on.

To paste contents from two files, you can do:

```
1 $ paste file1 file2
```

The syntax to use a different delimiter is as follows:

```
1 $ paste -d, file1 file2
```

Common delimiters are 'space', 'tab', '|', 'comma', etc.

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>cat phone
555-123-4567
555-231-3891
555-893-1048
555-732-7320
c7:/tmp>cat names
Wally
Beaver
Lumpy
June
c7:/tmp>paste phone names
555-123-4567    Wally
555-231-3891    Beaver
555-893-1048    Lumpy
555-732-7320    June
c7:/tmp>paste -d ':' phone names
555-123-4567:Wally
555-231-3891:Beaver
555-893-1048:Lumpy
555-732-7320:June
c7:/tmp>
```

join Command

To combine two files on a common field, at the command prompt type **join file1 file2** and press the ***Enter*** key.

For example, the common field (i.e. it contains the same values) among the phonebook and cities files is the phone number, and the result of joining these two files is shown in the screen capture.

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>cat phonebook
555-123-4567    Wally
555-231-3891    Beaver
555-893-1048    Lumpy
555-732-7320    June
c7:/tmp>cat cities
555-123-4567    Seattle
555-231-3891    Copenhagen
555-893-1048    Madison
555-732-7320    Corvallis
c7:/tmp>join phonebook cities
555-123-4567 Wally Seattle
555-231-3891 Beaver Copenhagen
555-893-1048 Lumpy Madison
555-732-7320 June Corvallis
c7:/tmp>
```

split Command

We will apply **split** to an American-English dictionary file of over 99,000 lines:

```
$ wc -l american-english
99171 american-english
```

where we have used **wc** (word count, soon to be discussed) to report on the number of lines in the file. Then, typing:

```
$ split american-english dictionary
```

will split the American-English file into 100 equal-sized segments named '**dictionaryxx**. The last one will, of course, be somewhat smaller.

```
student@ubuntu: /tmp
student@ubuntu:/tmp$ wc /usr/share/dict/american-english
 99171  99171 938848 /usr/share/dict/american-english
student@ubuntu:/tmp$ split /usr/share/dict/american-english dictionary
student@ubuntu:/tmp$ ls -l dictionary* | wc
    100     900    6100
student@ubuntu:/tmp$ ls -l dictionary* | head -10
-rw-rw-r-- 1 student student  8653 Dec 23 17:17 dictionaryaa
-rw-rw-r-- 1 student student  8552 Dec 23 17:17 dictionaryab
-rw-rw-r-- 1 student student  8876 Dec 23 17:17 dictionaryac
-rw-rw-r-- 1 student student  8842 Dec 23 17:17 dictionaryad
-rw-rw-r-- 1 student student  8249 Dec 23 17:17 dictionaryae
-rw-rw-r-- 1 student student  8405 Dec 23 17:17 dictionaryaf
-rw-rw-r-- 1 student student  8494 Dec 23 17:17 dictionaryag
-rw-rw-r-- 1 student student  8010 Dec 23 17:17 dictionaryah
-rw-rw-r-- 1 student student  8222 Dec 23 17:17 dictionaryai
-rw-rw-r-- 1 student student  8566 Dec 23 17:17 dictionaryaj
student@ubuntu:/tmp$
```

Regular Expressions and Search Patterns

Regular expressions are text strings used for matching a specific pattern, or to search for a specific location, such as the start or end of a line or a word. Regular expressions can contain both normal characters, as well as so-called meta-characters, such as ***** and **$**.

Many text editors and utilities such as **vi**, **sed**, **awk**, **find** and **grep** work extensively with regular expressions. Some of the popular computer languages that use regular expressions include Perl, Python and Ruby. It can get rather complicated, and there are whole books written about regular expressions; thus, we will do no more than skim the surface here.

These regular expressions are different from the wildcards (or meta-characters) used in filename matching in command shells such as bash. The table below lists search patterns and their usage.

Search Patterns	Usage
.(dot)	Match any single character
a|z	Match a or z
$	Match end of string
^	Match beginning of string
*****	Match preceding item 0 or more times

For example, consider the following sentence: *The quick brown fox jumped over the lazy dog.*

Some of the patterns that can be applied to this sentence are as follows:

Command	Usage
a..	matches **azy**
b.|j.	matches both **br** and **ju**
..$	matches **og**
l.*	matches **lazy dog**
l.*y	matches **lazy**
the.*	matches the whole sentence

grep Command

grep is extensively used as a primary text searching tool. It scans files for specified patterns and can be used with regular expressions, as well as simple strings, as shown in the table:

Command	Usage
grep [pattern] <filename>	Search for a pattern in a file and print all matching lines
grep -v [pattern] <filename>	Print all lines that do not match the pattern
grep [0-9] <filename>	Print the lines that contain the numbers 0 through 9
grep -C 3 [pattern] <filename>	Print context of lines (specified number of lines above and below the pattern) for matching the pattern; here, the number of lines is specified as 3

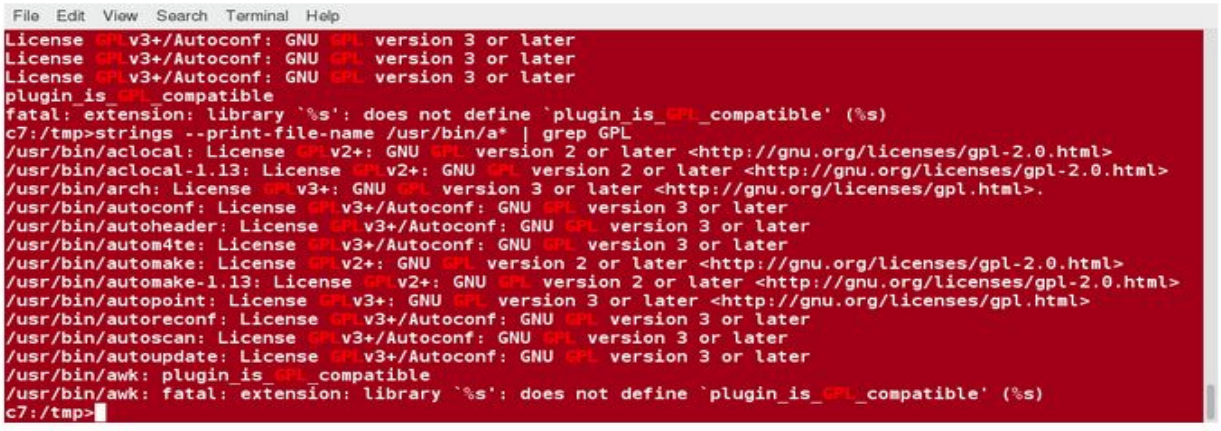
strings Command

strings is used to extract all printable character strings found in the file or files given as arguments. It is useful in locating human-readable content embedded in binary files; for text files, you can just use **grep**.

For example, to search for the string **my_string** in a spreadsheet:

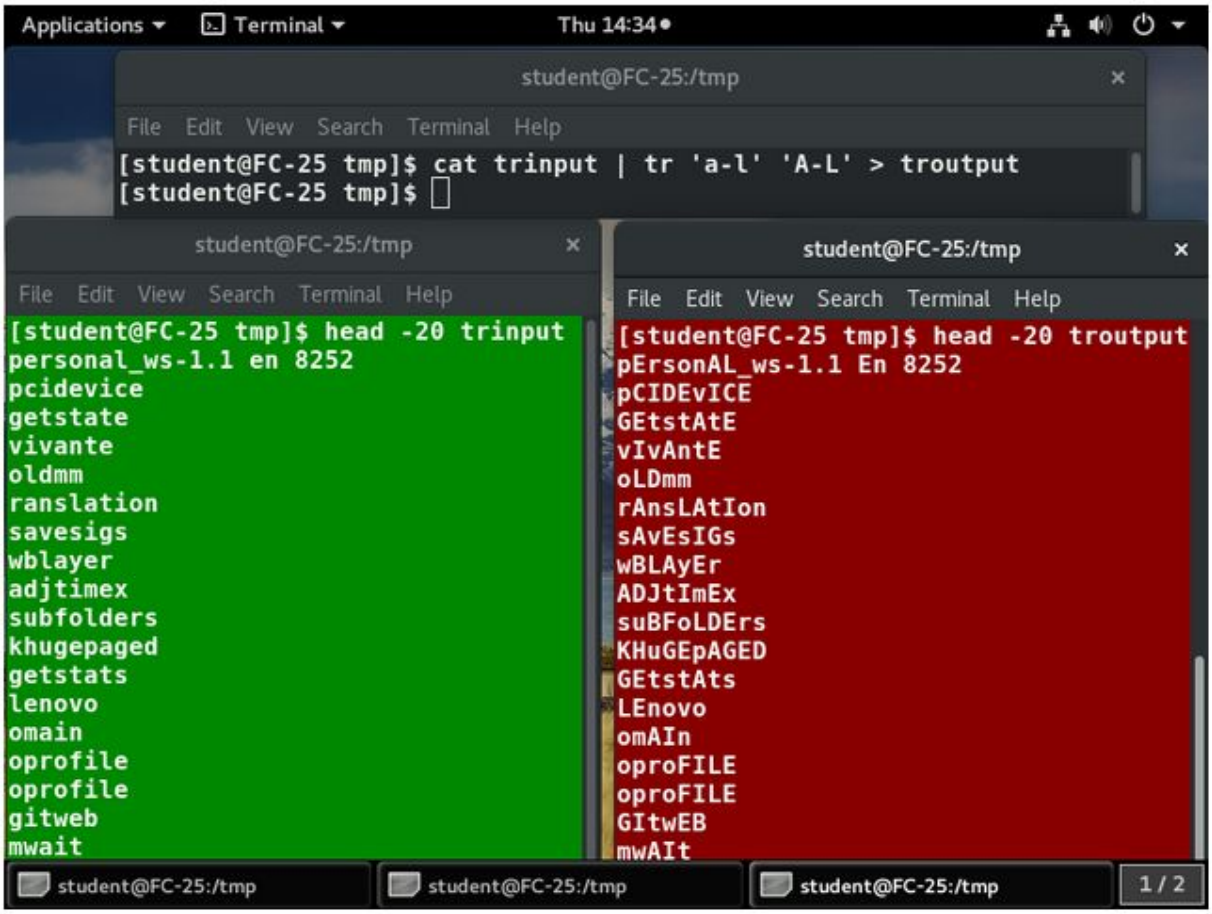
```
1  $ strings book1.xls | grep my_string
```

The screenshot below shows a search of a number of programs to see which ones have GPL licenses of various versions.



tr Command

Now, we will discuss some additional text utilities that you can use for performing various actions on your Linux files, such as changing the case of letters or determining the count of words, lines, and characters in a file.



The **tr** utility is used to translate specified characters into other characters or to delete them. The general syntax is as follows:

```
$ tr [options] set1 [set2]
```

The items in the square brackets are optional. **tr** requires at least one argument and accepts a maximum of two. The first designated **set1** in the example lists the characters in the text to be replaced or removed. The second, **set2**, lists the characters that are to be substituted for the characters listed in the first argument. Sometimes, these sets need to be surrounded by apostrophes (or single-quotes (')) in order to have the shell ignore that they mean something special to the shell. It is usually safe (and may be required) to use the single-quotes around each of the sets, as you will see in the examples below.

For example, suppose you have a file named **city** containing several lines of text in mixed case. To translate all lower case characters to upper case, at the command prompt type **cat city | tr a-z A-Z** and press the ***Enter*** key.

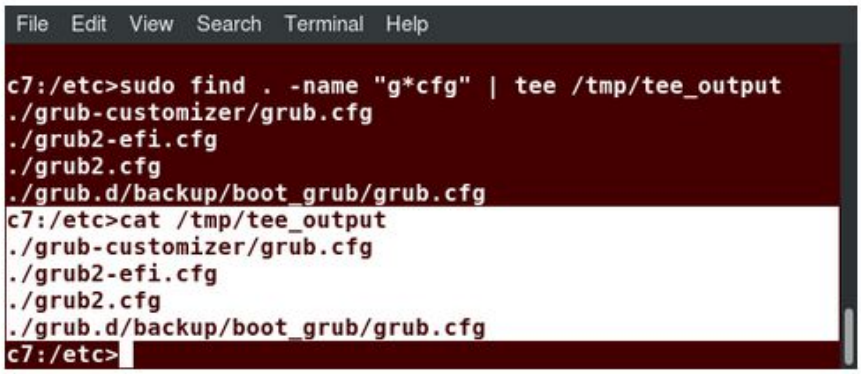
Command	Usage
$ tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	Convert lower case to upper case
$ tr '{}' '()' < inputfile > outputfile	Translate braces into parenthesis
$ echo "This is for testing" | tr [:space:] '\t'	Translate white-space to tabs
$ echo "This is for testing" | tr -s [:space:]	Squeeze repetition of characters using **-s**
$ echo "the geek stuff" | tr -d 't'	Delete specified characters using **-d** option
$ echo "my username is 432234" | tr -cd [:digit:]	Complement the sets using **-c** option
$ tr -cd [:print:] < file.txt	Remove all non-printable characters from a file
$ tr -s '\n' ' ' < file.txt	Join all the lines in a file into a single line

tee Command

tee takes the output from any command, and, while sending it to standard output, it also saves it to a file. In other words, it "tees" the output stream from the command: one stream is displayed on the standard output and the other is saved to a file.

For example, to list the contents of a directory on the screen and save the output to a file, at the command prompt type **ls -l | tee newfile** and press the ***Enter*** key.

Typing **cat newfile** will then display the output of **ls –l**.



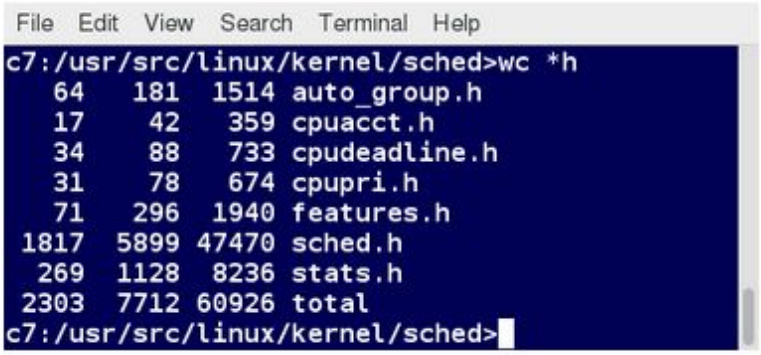
wc Command

wc (word count) counts the number of lines, words, and characters in a file or list of files. Options are given in the table below:

Option	Description
–l	Displays the number of lines
-c	Displays the number of bytes
-w	Displays the number of words

By default, all three of these options are active.

For example, to print only the number of lines contained in a file, type **wc -l filename** and press the ***Enter*** key.



cut Command

cut is used for manipulating column-based files and is designed to extract specific columns. The default column separator is the **tab** character. A different delimiter can be given as a command option.

For example, to display the third column delimited by a blank space, at the command prompt type **ls -l | cut -d" " -f3** and press the ***Enter*** key.

Bash Scripting

Script Basics

bash Script

After preparing a bash script and putting in a file, say **my_script.sh**, it can be invoked in either of two ways:

- Typing **bash my_script.sh**
- Have the first line of **my_script.sh** be **#!/bin/sh**
  - This will work with any other interpreter such as csh, perl, etc.
  - This is an exception to the rule that the character **#** is used to demark comments

Then make the script executable by doing **chmod +x my_script.sh** and then just running **./my_script.sh**

Environment Variables

- There are some special environment variables that can be used inside a script:
  - **$0** is the command name
  - **$1 $2** ... are the command arguments
  - **$*** represents them all
  - **$@** represents them all, preserving the grouping of quoted arguments
  - **$#** gives the number of arguments
- Note that you should often enclose these variables in double quotes; i.e. you should say "**$@**" to preserve the argument grouping
- In addition, you will get syntax errors when doing comparisons if the string is empty if you do not use double quotes

If **foobar.sh** is:

```
1  #!/bin/bash
2  echo 0 = $0
3  echo 1 = $1
4  echo '*' = $*
```

the output of **./foobar.sh a b c d e** is:

```
1  0 = ./foobar
2  1=a
3  *=abcde
```

Inside the script, the command **shift n** shifts the arguments **n** times (to the left).

There are two ways to include a script file inside another script:

- **. file**
- **source file**.

There are a number of options that can be used for debugging purposes:

- **set -n (bash -n)** just checks for syntax
- **set -x (bash -x)** echos all commands after running them
- **set -v (bash -v)** echos all commands before running them
- **set -u (bash -u)** causes the shell to treat using unset variables as an error
- **set -e (bash -e)** causes the script to exit immediately upon any non-zero exit status

where the **set** command is used inside the script (with a **+** sign behavior is reversed) and the second form, giving an option to **bash**, is invoked when running the script from the command line.

Conditionals

bash permits control structures like:

```
if condition
then
   statements
else
   statements
fi
```

There is also an **elif** statement.

Note that the variable **$?** holds the exit status of the previous command.

The condition can be expressed in several equivalent ways:

```
if [[ -f file.c ]] ; then ... ; fi
if [-ffile.c] ;then...;fi
if test -f file.c ; then ... ; fi
```

Remember to put spaces around the **[ ]** brackets.

The first form with double brackets is preferred over the second form with single brackets, which is now considered deprecated. For example, a statement such as:

```
if [ $VAR == "" ]
```

will produce a syntax error if **VAR** is empty, so you have to do:

```
if [ "$VAR" == "" ]
```

to avoid this.

The test form is also deprecated for the same reason and it is more clumsy as well. However, it is common to see these older conditional forms in many legacy scripts.

You will often see the **&&** and **||** operators (AND and OR, as in C) used in a compact shorthand:

```
$ make && make modules_install && make install
$ [[ -f /etc/foo.conf ]] || echo 'default config' >/etc/foo.conf
```

The **&&**s (ANDs) in the first statement say stop as soon as one of the commands fails; it is often preferable to using the **;** operator. The **||**' (ORs) in the second statement says stop as soon as one of the commands succeeds.

The **&&** operator can be used to do compact conditionals; e.g. the statement:

```
[[ $STRING == mystring ]] && echo mystring is "$STRING"
```

is equivalent to:

```
if [[ $STRING == mystring ]] ; then
    echo mystring is "$STRING"
fi
```

File Conditionals

There are many conditionals which can be used for various logical tests. Doing **man 1 test** will enumerate these tests. Grouped by category we find:

Test	Meaning
-e file	**file** exists?
-d file	**file** is a directory?
-f file	**file** is a regular file?
-s file	**file** has non-zero size?
-g file	**file** has **sgid** set?
-u file	**file** has **suid** set?
-r file	**file** is readable?
-w file	**file** is writeable?
-x file	**file** is executable?

String Comparisons

If you use single square brackets or the test syntax, be sure to enclose environment variables in quotes in the following:

Test	Meaning
string	**string** not empty?
string1 == string2	**string1** and **string2** same?
string1 != string2	**string1** and **string2** differ?
-n string	**string** not null?
-z string	**string** null?

Arithmetic Comparisons

These take the form:

```
1 exp1 -op exp2
```

where the operation **(-op)** can be:

```
1 -eq, -ne, -gt, -ge, -lt, -le
```

Note that any condition can be negated by use of **!**.

case

This construct is similar to the **switch** construct used in C code. For example:

```
#!/bin/sh
echo "Do you want to destroy your entire file system?"
read response

case "$response" in
   "yes")              echo "I hope you know what you are doing!" ;;
   "no" )              echo "You have some comon sense!" ;;
   "y" | "Y" | "YES" ) echo "I hope you know what you are doing!" ;
                       echo 'I am going to type: " rm -rf /"';;
   "n" | "N" | "NO" )  echo "You have some comon sense!" ;;
   *   )               echo "You have to give an answer!" ;;
esac
exit 0
```

Note the use of the **read** command, which reads user input into an environment variable.

Loops

bash permits several looping structures. They are best illustrated by examples.

for

```
for file in $(find . -name "*.o")
do
    echo "I am removing file: $file"
    rm -f "$file"
done
```

which is equivalent to:

```
$ find . -name "*.o" -exec rm {} ';'
```

or

```
$ find . -name "*.o" | xargs rm
```

showing use of the **xargs** utility.

while

```
#!/bin/sh

ntry_max=4 ; ntry=0 ; password=' '

while [[ $ntry -lt $ntry_max ]] ; do

   ntry=$(( $ntry + 1 ))
   echo -n 'Give password:  '
   read password
   if  [[ $password == "linux" ]] ; then
       echo "Congratulations: You gave the right password on try $ntry!"
       exit 0
   fi
   echo "You failed on try $ntry; try again!"
done

echo "you failed $ntry_max times; giving up"
exit -1
```

until

```
#!/bin/sh

ntry_max=4 ; ntry=0 ; password=' '

until [[ $ntry -ge $ntry_max ]] ; do

   ntry=$(( $ntry + 1 ))
   echo -n 'Give password:  '
   read password
   if [[ $password == "linux" ]] ; then
       echo "Congratulations: You gave the right password on try $ntry!"
       exit 0
   fi
   echo "You failed on try $ntry; try again!"

done

echo "you failed $ntry_max times; giving up"
exit -1
```

Functions

Functions

- bash permits use of subprograms or functions
- Alternatively, one script can call another script, but this can make passing information a little more complex, and increases the number of files; proper use of functions can make scripts much simpler
- The basic form of a function is:

```
fun_foobar(){
      statements
}
```

which will not look strange to any C programmer

A few things to remember about functions include:

- No arguments are enclosed in the parentheses
- Functions must always be defined **before** they are used, because scripts are interpreted, not compiled
- Functions leave positional arguments unchanged, but can reset other variables; a **`local var1 var2`** ... statement can make variables local

Example:

```
#!/bin/sh

test_fun1(){
    var=FUN_VAR
    shift
    echo "   PARS after fun shift:   $0 $1 $2 $3 $4 $5"
}

var=MAIN_VAR
echo ' '
echo "BEFORE FUN MAIN, VAR=$var"
echo "   PARS starting in main:  $0 $1 $2 $3 $4 $5"

test_fun1 "$@"
echo "   PARS after fun in main: $0 $1 $2 $3 $4 $5"
echo "AFTER FUN MAIN, VAR=$var"

exit 0
```

Sometimes you will see an (older) method of declaring functions, which explicitly includes a **function** keyword, as in:

```
function fun_foobar(){
   statements
}
```

or

```
function fun_foobar{
   statements
}
```

without the parentheses.

This syntax will work fine in **bash** scripts, but is not designed for the original **Bourne** shell, **sh**.

In the case where a function name is used which collides with an **alias**, this method will still work.

In most cases, use of the **function** keyword is not often used in new scripts.

Files and Filesystems

Types of Files

There are a limited number of basic file types, each of which are displayed in the following directory listing:

```
$ ls -lF
total 8
brw-r--r-- 1 coop coop 200, 0 Mar  9 15:07 a_block_device_node
crw-r--r-- 1 coop coop 200, 0 Mar  9 15:06 a_character_device_node
drwxrwxr-x 2 coop coop   4096 Mar  9 15:05 a_directory/
prw-rw-r-- 1 coop coop      0 Mar  9 15:04 a_fifo|
-rw-rw-r-- 1 coop coop   1601 Mar  9 15:04 a_file
lrwxrwxrwx 1 coop coop      4 Mar  9 15:06 a_symbolic_link_to_directory -> /usr/
lrwxrwxrwx 1 coop coop      6 Mar  9 15:10 a_symbolic_link_to_a_file -> a_file
srwxrwxr-x 1 coop coop      0 Mar  9 15:09 mysock=
```

The first character in the listing shows the type of file:

Character	Type
-	Normal File
d	Directory
l	Symbolic link
p	Named pipe (FIFO)
s	Unix domain socket
b	Block device node
c	Character device node

The **file** utility program can be used to ascertain file types, as in:

```
$ file *

acpitool:              ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV),
                           for GNU/Linux 2.6.9, dynamically linked
                           (uses shared libs), for GNU/Linux 2.6.9, not stripped
atob:                  C shell script text executable
awkit.sh:              Bourne-Again shell script text executable
blank:                 JPEG image data, JFIF standard 1.01
cdc.ORIG3:             Bourne-Again shell script text executable
copy_configs.sh:       ASCII text
gtk-recordMyDesktop:   python script text executable
gztobz2:               Bourne-Again shell script text executable
html2ps:               shell archive or script for antique kernel text
```

Types of Files

- Normal files (**-**) and directories (**d**) are the basic entries in the filesystem structure
- Symbolic links (**l**) point somewhere else on the system, such that a reference to either the link or its target is equivalent
- Named pipes (**p**), also called FIFOs, for First-In, First-Out, are used for inter-process communication
- Unix domain sockets (**s**) are used for a similar purpose, but do this through networking infrastructure, and have some more capabilities than FIFOs

- Block device nodes (**b**) and character device nodes (**c**) are used to communicate with either hardware devices, or software pseudo-devices
  - Block devices do reads and writes in predetermined fixed size chunks, and usually correspond to storage media such as disks and CD-ROMs; the I/O is generally buffered and cached
  - Character devices are addressed with streams of data, and correspond to devices such as sound cards, serial ports, etc; the I/O is generally not buffered or cached

  Device nodes are almost always placed in the **/dev** directory

Everything Is a File

- In describing UNIX-like operating systems, such as Linux, one often hears the old saying: *Everything is a file*
- This fits in with the philosophy that the same basic tools can be used for a wide variety of purposes, whether you are dealing with normal files, devices, etc., and can be strung together to accomplish non-trivial tasks.
- There are exceptions to this model; for example, network devices in Linux have no corresponding filesystem entry
- Note that in Linux file extensions rarely play a defining role; the type of a file is determined by examining its contents rather than its extension

Permissions and Access Rights

Access Rights

- When you do an **`ls -l as`** in:

  ```
  $ ls -l a_file
  -rw-rw-r-- 1 coop aproject 1601 Mar 9 15:04 a_file
  ```

  after the first character there are nine more which indicate the access rights granted to potential file users
- These are arranged in three groups of three:
  - owner: the user who owns the file (also called user)
  - group: the group of users who have access
  - world: the rest of the world (also called other)

  In the above listing, the user is **`coop`** and the group is **`aproject`**

read, write, execute

- Each of the triplets can have each of the following values set:
  - r: read access is allowed
  - w: write access is allowed
  - x: execute access is allowed
- If the permission is not allowed, a - appears instead of one of these characters
- Thus, in the preceding example,

  ```
  $ ls -l a_file
  -rw-rw-r-- 1 coop aproject 1601 Mar 9 15:04 a_file
  ```

  the user **`coop`** and members of the group **`aproject`** have read and write access, while anyone else has only read access

Changing Permissions and Access Rights

Changing file permissions is done with **chmod**, while changing file ownership is done with **chown**, and changing the group is done with **chgrp**.

There are a number of different ways to use **chmod**. For instance, to give the **owner** and **world** execute permission, and remove the **group** write permission:

```
$ ls -l a_file
-rw-rw-r-- 1 coop coop 1601 Mar  9 15:04 a_file
$ chmod uo+x,g-w a_file
$ ls -l a_file
-rwxr--r-x 1 coop coop 1601 Mar  9 15:04 a file
```

where **u** stands for user (owner), **o** stands for other (world), and **g** stands for group.

This kind of syntax can be difficult to type and remember, so one often uses a shorthand which lets you set all the permissions in one step. This is done with a simple algorithm, and a single digit suffices to specify all three permission bits for each entity. This digit is the sum of:

- 4 if read permission is desired
- 2 if write permission is desired
- 1 if execute permission is desired.

Thus, 7 means read/write/execute, 6 means read/write, and 5 means read/execute.

When you apply this when using **chmod**, you have to give three digits for each degree of freedom, such as in:

```
$ chmod 755 a_file
$ ls -l a_file
-rwxr-xr-x 1 coop coop 1601 Mar  9 15:04 a file
```

Changing the group ownership of the file is as simple as doing:

```
$ chgrp aproject a_file
```

and changing the ownership is as simple as:

```
$ chown coop a_file
```

You can change both at the same time with:

```
$ chown coop.aproject a_file
```

where you separate the owner and the group with a period.

All three of these commands can take an **-R** option, which stands for recursive. For example:

```
$ chown -R coop.aproject .
$ chown -R coop.aproject subdir
```

will change the owner and group of all files in the current directory and all its subdirectories in the first command, and in **subdir** and all its subdirectories in the second command.

These commands can only do what you already have the right to do. You cannot change the permissions on a file you do not own, for example, or switch to a group you are not a member of. To do such changes, you must be root, prefixing **sudo** to most of the above commands.

Linux Filesystems

Virtual File System (VFS)

- Linux implements a **Virtual File System** (**VFS**), as do all modern operating systems
- For the most part neither the specific filesystem or actual physical media and hardware need be addressed by filesystem operations
- Furthermore, network filesystems (such as NFS) can be handled transparently
- This permits Linux to work with more filesystem varieties than any other operating system
- This democratic attribute has been a large factor in its success
- Most filesystems have full read/write access while a few have only read access and perhaps experimental write access

- Some filesystem types, especially non-UNIX based ones, may require more manipulation in order to be represented in the VFS
- For example, variants such as vfat do not have distinct read/write/execute permissions for the owner/group/world fields; the VFS has to make an assumption about how to specify distinct permissions for the three types of user, and such behavior can be influenced by mounting operations
- Even more drastically, such filesystems store information about files in a File Allocation Table (FAT) at the beginning of the disk, rather than in the directories themselves, a basically different architectural method
- A number of newer high performance filesystems include full journaling capability

ext2, ext3, and ext4 Filesystems

- The native filesystems for Linux are **ext2** and its journaling descendants, **ext3** and **ext4**
- Each has its associated utility for formatting the filesystem (e.g. mkfs.ext3) and for checking the filesystem (e.g. fsck.ext4)
- Additionally, many parameters can be reset or tuned after filesystem creation with the program tune2fs; while defragmentation is generally not necessary, one can use e4defrag to do so
- The ext4 filesystem had its experimental designation removed with the 2.6.28 kernel release
- Pre-existing ext3 partitions could be migrated in place to ext4 without risk; with new features to be used only in subsequent file system operations
- One immediately obvious improvement is fast fsck; the speed of a filesystem check can easily go up by an order of magnitude or more

Journaling Filesystems

- Journaling filesystems recover from system crashes or ungraceful shutdowns with little or no corruption, and they do so very rapidly
- While this comes at the price of having some more operations to do, additional enhancements can more than offset the price
- In a journaling filesystem, operations are grouped into transactions
- A transaction must be completed without error, atomically; otherwise the filesystem is not changed
- A log file is maintained of transactions; when an error occurs, usually only the last transaction needs to be examined

Journaling Filesystems Freely Available Under Linux

The following journaling filesystems are freely available under Linux:

- The **ext3** filesystem is an extension of the ext2 filesystem
- The **ext4** filesystem is an extension of the ext3 filesystem, and was included in the mainline kernel first as an experimental branch, and then as a stable production feature in kernel 2.6.28 (features: extents, 48-bit block numbers, and up to 16 TB size)
- The **Reiser** filesystem was the first journaling implementation in Linux, but has lost its leadership and development has stalled
- The **JFS** filesystem is a product of IBM; has been ported from IBM's AIX OS
- The **XFS** filesystem is a product of SGI; has been ported from SGI's IRIX OS
- The **btrfs** filesystem (**B TR**ee filesystem) is the most recent, is native to Linux, and has many advanced features

btrfs

- Both Linux developers and Linux users with high performance and high capacity or specialized needs are following the development and gradual deployment of the **btrfs** filesystem, which was created by Chris Mason
- While btrfs has been in the mainline kernel since 2.6.29, it has generally been viewed as experimental; although it has been used in new products
- One of the main features is the ability to take frequent snapshots of entire filesystems, or sub-volumes of entire filesystems in virtually no time
- Because btrfs makes extensive use of **COW** (Copy on Write) techniques , such a snapshot does not involve any more initial space for data blocks or any I/O activity except for some metadata updating

- One can easily revert to the state described by earlier snapshots and even induce the kernel to reboot off an earlier root filesystem snapshot
- btrfs maintains its own internal framework for adding or removing new partitions and/or physical media to existing filesystems, much as LVM (Logical Volume Management) does
- Before btrfs becomes suitable for day-to-day use in critical filesystems, some tasks require finishing; for example, as of now the fsck (filesystem checking) program is read-only, which means you can find out what is wrong, but you cannot easily fix it!

Mounting Filesystems

In UNIX-like operating systems, all files are arranged in one big filesystem tree rooted at **/**. Many different partitions on many different devices may be coalesced together by mounting partitions on various mount points, or directories in the tree.

The full form of the **mount** command is:

```
1 $ sudo mount [-t type] [-o options] device dir
```

In most cases, the filesystem type can be deduced automatically from the first few bytes of the partition, and default options can be used, so it can be as simple as:

```
1 $ sudo mount /dev/sda8 /usr/local
```

Most filesystems need to be loaded at boot and the information required to specify mount points, options, devices, etc., is specified in **/etc/fstab**:

File Edit View Search Terminal Help

```
c7:/tmp>cat /etc/fstab
#
# /etc/fstab
# Created by anaconda on Thu Jan 15 19:25:00 2015
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
LABEL=RHEL7     /           ext4       defaults        1 1
LABEL=local     /usr/local  ext4  defaults 1 2
LABEL=src       /usr/src    ext4  defaults 1 2
LABEL=pictures  /PICTURES   ext4  defaults 1 2
LABEL=dead      /DEAD       ext4  defaults 1 2
LABEL=dead2     /DEAD2      ext4  defaults 1 2
LABEL=virtual   /VIRTUAL    ext4  defaults 1 2
LABEL=iso_images /ISO_IMAGES ext4  defaults 1 2
LABEL=audio     /AUDIO      ext4  defaults 1 2
LABEL=vms       /VMS         ext4  defaults 1 2
/usr/src/KERNELS.sqfs /usr/src/KERNELS squashfs loop 0 0

LABEL=SWAP swap                    swap    defaults        0 0
#UUID=471dfeba-3ec7-4529-8069-2afe50762c57 /  ext4   defaults  1 1
c7:/tmp>
```

Note that in this example, most of the filesystems are mounted by **label**; it is also possible to mount by device name or **UUID**; the following are all equivalent:

```
$ sudo mount /dev/sda2   /boot
$ sudo mount LABEL=boot  /boot
$ sudo mount    -L boot  /boot
$ sudo mount UUID=26d58ee2-9d20-4dc7-b6ab-aa87c3cfb69a /boot
$ sudo mount   -U 26d58ee2-9d20-4dc7-b6ab-aa87c3cfb69a /boot
```

The list of currently mounted filesystems can be seen with:

```
$ sudo mount
/dev/sda5 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda6 on /RHEL6-32 type ext3 (rw)
/dev/mapper/VGN-local on /usr/local type ext4 (rw)
/dev/mapper/VGN-tmp on /tmp type ext4 (rw)
/dev/mapper/VGN-src on /usr/src type ext4 (rw)
/dev/mapper/VGN-virtual on /VIRTUAL type ext4 (rw)
/dev/mapper/VGN-beagle on /BEAGLE type ext4 (rw)
tmpfs on /dev/shm type tmpfs (rw)
debugfs on /sys/kernel/debug type debugfs (rw)
/dev/sda1 on /c type fuseblk (rw,allow_other,default_permissions,blksize=4096)
/usr/local/teaching/FTP/LFT on /var/ftp/pub2 type none (rw,bind)
/ISO_IMAGES/CENTOS/CentOS-5.5-x86_64-bin-DVD-1of2.iso on /var/ftp/pub
                              type iso9660 (rw,loop=/dev/loop0)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/sda2 on /boot type ext3 (rw)
```

If a directory is used as a mount point, its previous contents are hidden under the newly mounted filesystem. A given partition can be mounted in more than one place and changes are effective in all locations.

You can also mount NFS (Network File Systems) as in:

```
$ sudo mount 192.168.1.100:/var/ftp/pub /mnt
```

RAID and LVM

Redundant Array of Independent Disks (RAID)

- RAID (Redundant Array of Independent Disks) spreads I/O over multiple spindles, or disks; this can really increase performance in modern disk controller interfaces, such as SCSI which can perform the work in parallel efficiently
- RAID can be implemented either in software (it is a mature part of the Linux kernel) or in hardware
- If your hardware RAID is known to be of good quality it should be more efficient than using software RAID
- With the hardware implementation the operating system is actually not directly aware of using RAID; it is transparent
    - For example, three 512 GB hard drives (two for data, one for parity) configured with RAID-5 will just look like a single 1 TB disk

Features of RAID

- Three essential features of RAID are:
  - **Mirroring**: writing the same data to more than one disk
  - **Striping**: splitting of data to more than one disk
  - **Parity**: extra data is stored to allow problem detection and repair, yielding fault tolerance
- Thus use of RAID can improve both performance and reliability
- There are a number of RAID specifications of increasing complexity and use; the most commonly used are levels 0, 1, and 5

Logical Volume Management (LVM)

- Using **LVM** (**L**ogical **V**olume **M**anagement) breaks up one virtual partition into multiple chunks, which can be on different physical volumes
- There are many advantages to using LVM; in particular it becomes really easy to change the size of the logical partitions and filesystems, to add more space, rearrange things, etc.
- One or more physical disk partitions, or **physical volumes**, are grouped together into a **volume group**
- Then, the volume group is subdivided into logical volumes which appear to the system as disk partitions and are then formatted to contain mountable filesystems

- There are a variety of command line utilities tasked to create, delete, resize, etc. physical and logical volumes; fortunately, these command line utilities are not hard to use and are quite flexible
- Performance changes do occur
- There is a definite additional cost that comes from the overhead of the LVM layer; however, even on non-RAID systems, if you use striping in the setup you can achieve some parallelization improvements
- Such striping does no good if it is within the same physical disk, however

gcc and Other Compilers

gcc

- **gcc** is the **G**NU **C** compiler; its proper name is the **G**NU **C**ompiler **C**ollection
- It can be invoked as gcc or cc, and can compile programs written in C, C++, and Objective C
- **g++** is the C++ compiler; it can also be invoked as c++
- gcc works closely with the GNU libc, **glibc**, and the debugger, **gdb**
- Virtually every operating system you can think of has a version of gcc, and it can be used for cross-compilation on different architectures

- gcc also forms the back end for compiler front ends in Ada95 (package gcc-gnat), Fortran (package gcc-gfortran), and Pascal (package gcc-gpc); i.e. first there is a translation to the C language, and then a back end (silently) invokes gcc
- The gcc-java package supplies gcj which adds support for compiling Java programs and bytecode into native code, but is no longer available on some recent Linux distributions as it is considered obsolete

Compiling Stages

- Invoking gcc actually entails a number of different programs or stages, each of which has its own **man** page, and can be independently and directly invoked; these are:

Stage	Command	Default Input	Default Output	**-W** Switch
Preprocessing	cpp	`.c`	`.i`	`-Wp....`
Compilation	gcc	`.i`	`.s`	N/A
Assembly	as	`.s`	`.o`	`-Wa....`
Linking	ld	`.o`	`a.out`	`-Wl....`

- Depending on your Linux distribution, details about the gcc installation and defaults can be found in the **`/usr/lib/gcc`**, **`/usr/lib64/gcc`** and/or **`/usr/libexec/gcc`** directories

LLVM

- The **LLVMLinux** project provides a new alternative compiler, meant to be used for both user applications and the Linux kernel
- It is rapidly reaching maturation for compiling the Linux kernel (which is rather idiosyncratic in its use of specialized gcc options) and is already used in production code for applications

Intel

- **Intel** has a mature set of compilers
- Evaluation copies can be downloaded for free, and a free non-commercial license can be obtained for learning purposes
- The Intel C compiler works well for compiling applications under Linux; it can be used to compile the kernel but it is not a trivial exercise and it is doubtful anyone is using it for this purpose in a production environment

Major gcc Options

The compiled code format will be **ELF** (Executable and Linkable Format), which makes using shared libraries easy; the older **a.out** format, while obsolete (although the name **a.out** survives, confusingly, as the default name for an output file), may still be used if the Linux kernel has been configured to support it.

Here is a list of some of the main options that can be given to **gcc**:

Compiler Path Options

Compiler Path Options

Option	Description
-I dir	Include **dir** in search for included files; cumulative
-L dir	Search **dir** for libraries; cumulative
-l	Link to **lib**; **-lfoo** links to **libfoo.so** if it exists, or to **libfoo.a** as a second choice

Compiler Preprocessor Options

Option	Description
-M	Do not compile; give dependencies for make
-H	Print out names of included files
-E	Preprocess only
-D def	Define **def**
-U def	Undefine **def**
-d	Print **#defines**

Compiler Warning Options

Option	Description
-v	Verbose mode, gives version number
-pedantic	Warn very verbosely
-w	Suppress warnings
-W	More verbose warnings
-Wall	Enable a bunch of important warnings

Compiler Debugging and Profiling Options

Option	Description
-g	Include debugging information
-pg	Provide profile information for **gprof**

Compiler Input and Output Options

Option	Description
-c	Stop after creating object files, do not link
-o file	Output is file ; default is **a.out**
-x lang	Expect input to be in **lang**, which can be **c**, **objective-c**, **c++** (and some others); otherwise, guess by input file extension

Compiler Control Options

Option	Description
-ansi	Enforce full ANSI compliance
-pipe	Use pipes between stages
-static	Suppress linking with shared libraries
-O[lev]	Optimization level; 0, 1, 2, 3; default is 0
-Os	Optimize for size; use all **-O2** options except those that increase the size

A good set of options to use is:

```
1 -O2 -Wall -pedantic
```

Make sure you understand any warnings; if you take the effort to obliterate them, you might save yourself a lot of debugging. However, do not use **-pedantic** when compiling code for the Linux kernel, which uses many **gcc** extensions.

Static Libraries

Static libraries have the extension **.a**. When a program is compiled, full copies of any loaded library routines are incorporated as part of the executable.

The following tools are used for maintaining static libraries:

- **ar** creates, updates, lists and extracts files from the library. The command:

```
1 $ ar rv libsubs.a *.o
```

will create **libsubs.a** if it does not exist, and insert or update any object files in the current directory.

- **ranlib** generates, and stores within an archive, an index to its contents. It lists each symbol defined by the relocatable object files in the archive. This index speeds up linking to the library. The command:

```
$ ranlib libsubs.a
```

is completely equivalent to running **ar -s libsubs.a**. While running **ranlib** is essential under some UNIX implementations, under Linux it is not strictly necessary, but it is a good habit to get into.

- **nm** lists symbols from object files or libraries. The command:

```
$ nm -s libsubs.a
```

gives useful information. **nm** has a lot of other options.

Modern applications generally prefer to use shared libraries, as it is more efficient and conserves memory. However, there are at least two circumstances where static libraries are still used:

1. For programs that are used early in system startup, before the tools to work with shared libraries are fully operational.
2. For programs that want to be completely self-contained and not have to deal with potential problems from system updates of libraries that are utilized by the application. This is typically done by either proprietary (and even closed source) application suppliers, or by other large vendors, such as Google or Mozilla. This is always controversial, because it is up to the vendor to make sure that any security holes or other bugs are fixed when they are discovered upstream in the included libraries.

Shared Libraries

A single copy of a shared library can be used by many applications at once; thus, both executable sizes and application load time are reduced.

Shared libraries have the extension **.so**. Typically, the full name is something like **libc.so.N** where **N** is a major version number.

Under Linux, shared libraries are carefully versioned. For example, a shared library might have any of the following names:

- **libmyfuncs.so.1.0** - The actual shared library.
- **libmyfuncs.so.1** - The name included in the **soname** field of the library. Used by the executable at runtime to find the latest revision of the **v. 1myfuncs** library.
- **libmyfuncs.so** - Used by **gcc** to resolve symbol names in the library at link time when the executable is created.

To create a shared library, first you must compile all sources with the **-fPIC** option, which generates so-called Position Independent Code; do not use **-fpic**; it produces somewhat faster code on **m68k**, **m88k**, and **Sparc** chips, but imposes arbitrary size limits on shared libraries.

```
$ gcc -fPIC -c func1.c
$ gcc -fPIC -c func2.c
```

To create a shared library, the option **-shared** must be given during compilation, giving the **soname** of the library, as well as the full library name as the output:

```
$ gcc -fPIC -shared -Wl,-soname=libmyfuncs.so.1 *.o -o libmyfuncs.so.1.0 -lc
```

where the **-Wl** tells **gcc** to pass the option to the linker. The **-lc** tells the linker that **libc** is also needed, which is generally the case.

You can usually get away ignoring the **-fPIC** and **-Wl**, **soname** options, but it is not a good idea. This is because **gcc** normally emits such code anyway. In fact, giving the option prevents the compiler from ever issuing position-dependent code.

Note that it is really the linker (**ld**) that is doing the work, and the above step could also have been written as:

```
$ ld -shared -soname=libmyfuncs.so.1 *.o -o libmyfuncs.so.1.0 -lc
```

To get the above to link and run properly, you will also have to do:

```
$ ln -s libmyfuncs.so.1.0 libmyfuncs.so
$ ln -s libmyfuncs.so.1.0 libmyfuncs.so.1
```

If you leave out the first symbolic link, you will not be able to compile the program; if you leave out the second, you will not be able to run it. It is at this point that the version check is done, and if you do not use the **-soname** option when compiling the library, no such check will be done.

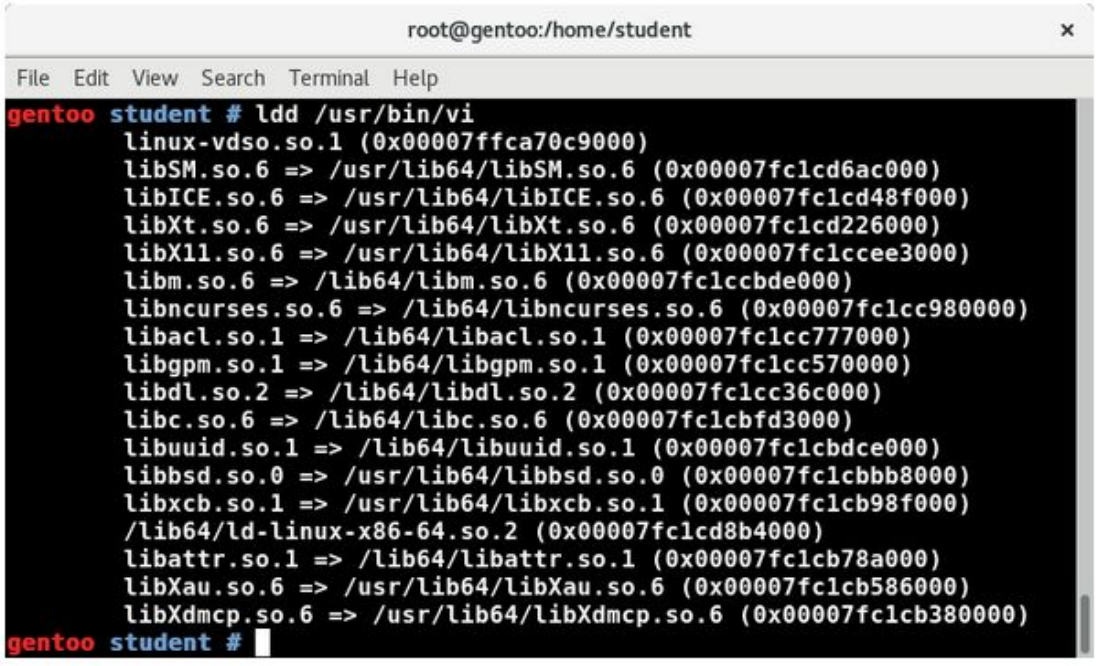
In many cases, both a shared and a static version of the same library may exist on the system, and the linker will choose the shared version by default. This may be overridden with the **-static** option to the linker. However, if you do this, all libraries will be linked in statically, including **libc**, so the resulting executable will be large; you have to be more careful than that.

The GNU **libtool** script can assist in providing shared library support, helping with compiling and linking libraries and executables, and is particularly useful for distributing applications and packages. Full documentation can be obtained by doing **info libtool**.

Finding shared libraries

A program which uses shared libraries has to be able to find them at run time.

ldd can be used to ascertain what shared libraries an executable requires. It shows the **soname** of the library and what file it actually points to:



ldconfig is generally run at boot time (but can be run anytime), and uses the file **/etc/ld.so.conf**, which lists the directories that will be searched for shared libraries. **ldconfig** must be run as root and shared libraries should only be stored in system directories when they are stable and useful.

Besides searching the data base built up by **ldconfig**, the linker will first search any directories specified in the environment variable **LD_LIBRARY_PATH**, a colon separated list of directories, as in the **PATH** variable. So, you can do:

```
$ LD_LIBRARY_PATH=$HOME/foo/lib
$ foo [args]
```

or

```
$ LD_LIBRARY_PATH=$HOME/foo/lib  foo [args]
```

Linking to Libraries

Whether a library is static or fixed, executables are linked to the library with:

```
$ gcc -o foo foo.c -L/mypath/lib -lfoolib
```

This will link in **/mypath/lib/libfoolib.so**, if it exists, and **/mypath/lib/libfoolib.a** otherwise.

The name convention is such that **-lwhat** refers to library **libwhat.so(.a)**. If both **libwhat.so** and **libwhat.a** exist, the shared library will be used, unless **-static** is used on the compile line.

Poorly designed projects may have circular library dependencies, and since the loader makes only one pass through the libraries requested, you can have something like:

```
$ gcc .... -lA -lB -lA ..
```

if **libB** refers to a symbol in **libA** which is not otherwise referred to. While there is an option to make the loader make multiple passes (see info **gcc** for details) it is very slow, and a proper, layered, library architecture should avoid this kind of going in circles.

The default library search path will always include **/usr/lib** and **/lib**. To see exactly what is being searched you can do **gcc --print-search-dirs**. User-specified library paths come before the default ones, although there are extended options to **gcc** to reverse this pattern.

Stripping executables

The command:

```
$ strip foobar
```

where **foobar** is an executable program, object file, or library archive, can be used to reduce file size and save disk space. The symbol table is discarded. This step is generally done on production versions.

Do not use **strip** on either the Linux kernel or kernel modules, both of which need the symbol information!

Getting Debug Information

You can use the environment variable **LD_DEBUG** to obtain useful debugging information. For instance, doing:

```
$ LD_DEBUG=help
```

and then typing any command gives:

```
$ ls
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
$
```

You can also show how symbols are resolved upon program execution, and help find things like linking to the wrong version of a library. Try:

```
$ LD_DEBUG=all  ls
```

to see a sample.

Debugging with gdb

gdb

- **gdb** is the GNU debugger
- Upon launch, after processing all command line arguments and options, it loads commands from the file **`.gdbinit`** in the current working directory (if it exists)
- gdb allows you to step through C and C++ programs, setting breakpoints, displaying variables, etc. (actually it will work with programs in native Fortran and other languages that use gcc as a back end as well)
- In addition, gdb can properly debug multi-threaded programs
- Programs have to be compiled with the **`-g`** option for symbol and line number information to be available to gdb
- Note, however, you can still use gdb to get some information even if this has not been done; for instance, the **where** command often tells you exactly where the program bombed

Graphical Interfaces

- There are a number of graphical interfaces to gdb which may make it easier to use; we will mention only:
  - Larger **integrated development environments** (IDE) such as **Eclipse** are often used as a graphical debugging interface
  - **ddd** may be offered by your distribution as a standard package; while it does not provide a complete integrated development environment, it can easily provide most of the ingredients for building your own
- Using a heavier and well-supported complete IDE (such as Eclipse) is now often preferred
- No matter what graphical interface you use, the debugger remains gdb

Java Installation and Environment

JAVA: Write Once, Use Anywhere?

- The concept of *Write Once, Run Anywhere* (WORA) was promoted by Sun to highlight the cross-platform capabilities of the **Java** language; sometimes the phrase *Write Once, Run Everywhere* (WORE) is also used
- Theoretically, at least, Java code can be developed on any platform that has a **JVM** (Java Virtual Machine)
- Once it has been compiled into bytecode it is expected to run on any hardware from a cell phone to a mainframe, and on any operating system, without requiring further fiddling
- The reality is that there are multiple JVM implementations available, such as those from Oracle/Sun, openjdk, and IBM
- Furthermore, there are many operating systems, of which Linux is only one, and within Linux itself there are many distributions and each has multiple versions

- Thus, if you really want to make sure things will behave properly everywhere, you have to check for the entire matrix of JVM and operating system combinations; this has led to the joke that the concept should really be called *Write Once, Debug Everywhere*
- However, this stance is too gloomy for at least three reasons:
  - JVM implementations have matured - standardized compliance is now very good
  - The actual choice of platforms an application will have to run on is far more limited in actual practice than the large number of entries in the full matrix, and many entries in the matrix are indistinguishable in practice
  - The advantage of the Java abstraction layer is so high that not having to recompile for every possible architecture and operating system outweighs potential problems
- As a rule, WORA has worked better on the server side than it has on the client side because of the difficulties of dealing with human interface standards and desktop/window managers and operating systems

It is possible that you will have (or want to have) multiple versions of Java components installed on your system. While you can directly invoke a specific choice without disturbing the general system configuration, it is also possible to set the system default in an easily reversible way.

In order to enable this, most Linux systems use the alternatives tool to set the system default. This utility is used for many alternative setting tasks on the system, not just those which are Java-related.

For example, to reconfigure the choice for Java on a Red Hat-based system:

```
$ sudo alternatives --config java

There are 2 programs which provide 'java'.

  Selection    Command
-----------------------------------------------
   1           java-1.7.0-openjdk.x86_64 \
                      (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.161-2.6.12.0.el7_4
                        .x86_64/jre/bin/java)
*+ 2           java-1.8.0-openjdk.x86_64 \
                      (/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.161-0.b14.el7_4
                        .x86_64/jre/bin/java)
Enter to keep the current selection[+], or type selection number:
```

How to use this is pretty obvious.

On Debian-based systems, such as Ubuntu, the command is **update-alternatives**.

How this works is pretty straightforward. The directory **/etc/alternatives** contains symbolic links to the proper location:

```
$ ls -l /etc/alternatives/java

lrwxrwxrwx 1 root root 73 Jan 19 07:02 /etc/alternatives/java -> \
     /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.161-0.b14.el7_4.x86_64/jre/bin/java
```

Note that the generic Java binary is also just a symbolic link:

```
$ which java
/usr/bin/java
$ ls -l /usr/bin/java
lrwxrwxrwx 1 root root 22 Jan 19 07:02 /usr/bin/java -> /etc/alternatives/java
```

Package Management

Why Use Package Management?

- Originally, most Linux distributions were simply collections of **tarballs**, of either binaries or sources which required compilation; some distributions (**Slackware**) still work this way
- However, this method has many disadvantages:
  - Removing all files from a package can be difficult
  - One might accidentally delete a package that other packages need, or install a package that will not work because it needs other packages that have not been installed
  - A developer may lose track of what exact sources were used to build a particular binary package
  - Updates and upgrades can be very difficult, for a number of reasons:
    - Files which are no longer needed may remain on the system
    - In place upgrades done while the system is running may cause difficulties, even system crashes
    - The order of upgrading software packages may be important but not be explicitly considered
    - Software groups that require simultaneous updating may conflict with each other

RPM and APT

- Several systems have been developed in the attempt to improve on this process, these include:
  - **RPM** (Red Hat Package Manager): originally only Red Hat used RPM, but many other Linux distributors have used RPM, such as Fedora, and particularly SUSE and its derivatives, such as OpenSUSE
  - **APT** (Advanced Packaging Tool): produces deb packages; originally developed by Debian it is the foundation of Ubuntu and other Debian-derived distributions such as Linux Mint
- **alien** is a tool for converting back and forth between rpm and deb packages, or installing one kind of package on a system set up for the other; however, alien is not available on all Linux distributions

Packaging System Benefits for Developers

- Repeatable builds
- Generation of dependency data, such as what other packages are needed by a given package, and/or what other packages (or class of packages) may need a given package
- Inclusion of the pristine sources, which aids in configuration control and establishing a clear history of revision
- Full explanation of any patches that have been made to the upstream sources, together with instructions for how to proceed in building the package as well as in installing it

Packaging System Benefits for System Administrators and End Users

- Integrity of the installation can be verified in a uniform and rapid fashion
- Simple installation and removal methods
- Package updates and upgrades automatically preserve customized and modified configuration files
- Error-checking on install and remove (erase) ensures needed resources are not obliterated
- Users and administrators can do queries on matters including identifying what files are part of a package, or the inverse process of asking what package (if any) a given file is part of

Maintenance

- Maintenance of software by using packaging systems is an essential task of any Linux distribution
- Every other function that must be accomplished requires coherent packaging, including:
  - Keeping a healthy bi-directional connection to the upstream developers, both incorporating patches from upstream as well as sending patches to the upstream maintainers
  - Establishing and deploying clear policies for dealing with configuration files, etc., as packages are updated
  - Updating and upgrading packages in timely fashion, both to fix outstanding bugs and security holes, and to incorporate new features
  - Most importantly, perhaps, ensuring proper package dependencies, including control the order of installation and removal, requiring various packages to be dealt with as one group, etc.
  - Maintaining (multiple) repositories, ensuring that they are secure and authoritative, as well as complete

- Linux distributions include software from many sources
- Maintaining coherence, ensuring proper licensing compliance, etc., is actually a very difficult task.
- The package maintainer may be an employee of the distribution (especially for commercial distributions such as Red Hat or Ubuntu) or a "volunteer" for other distributions such as Debian or Fedora
- In either case, the maintainer has to think long-term, how to really do a good job on the package building so that updates, upgrades, use of different systems, can be done rather cleanly with time

RPM Creation

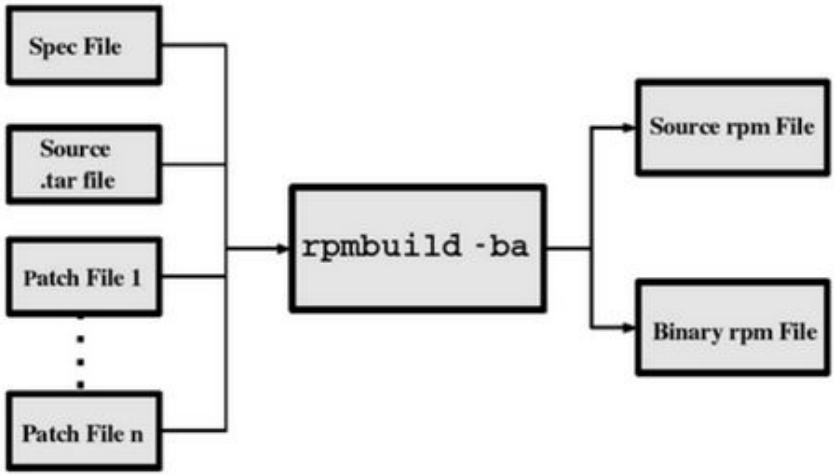
The RPM packaging system separates its main functionality methods using two main programs:

- **rpm**: Handles querying, installing, upgrading and erasing.
- **rpmbuild**: Handles creating and manipulating source and binary packages.

Each of these programs comes packaged as part of its own **rpm** and comes together with other related utilities we will not discuss in this section:

```
$ rpm -ql rpm-build | grep bin
/usr/bin/gendiff
/usr/bin/rpmbuild
/usr/bin/rpmspec
$ rpm -ql rpm | grep bin
/bin/rpm
/usr/bin/rpm2cpio
/usr/bin/rpmdb
/usr/bin/rpmkeys
/usr/bin/rpmquery
/usr/bin/rpmverify
```

It is a rather obvious point, but you should never remove the **rpm** program itself!



There are three basic ingredients required to assemble the source and binary packages:

- A **tarball** file containing all source code, makefiles, default configuration files, documentation, etc.
- Any **patch** files needed to be applied to the upstream source encapsulated in the tarball.
- A **spec** file that must be written that describes the package and how to patch and build it, dependency information, etc.

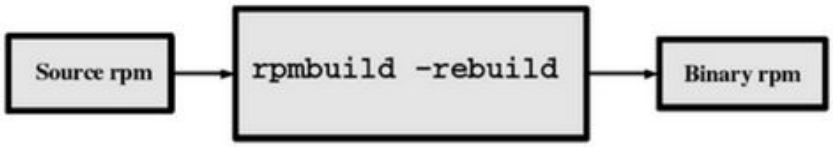
Once you have all these files together, the command:

```
$ rpmbuild -ba specFile
```

will perform the following steps:

- Coalesce the input files into the source RPM.
- Unpack the source tar file.
- Apply all patch files.
- Build binaries for the current architecture.
- Package binaries and config files into a binary RPM file.

Each of these steps is described by the **spec** file that the packager wrote.



RPM can be picky about where these files are placed. We will return to this question in the laboratory exercise.

If you want binaries for multiple platforms, then once you have the source RPM you can just copy it to a machine of another architecture and run **rpmbuild --rebuild SourceRPM** (or you can do a cross-compile if you have the cross-compile toolchain installed). This command can also be used to re-create the binary RPM from the source RPM in case you ever need to do so.

Also note that some "binary" rpms contain no binary files, i.e. they contain only scripts, configuration files, etc., that work independently of architecture. They will have the string **noarch** in their name rather than **i686 or x86_64**, etc., in their binary package name.

RPM Spec File

A **spec** file is a collection of package information and shell scripts. Many of of the shell scripts are very short (one line scripts are common). Each script performs one of the tasks necessary in building, installing, or uninstalling a package. The main sections of the file are:

Section Name	Required?	Section Type	Purpose
Header (start of file)	Y	Data fields	Provide the rpm -i data
%description	Y	ASCII text	Provide a text description of the package
%prep	Y	Shell script	Unpack source code and apply patches
%changelog	Y	Change history	History of changes
%build	Y	Shell script	Build the binary from the unpacked sources
%install	Y	Shell script	Copy the binaries to their destination
%files	Y	List	Every file that the package provides is listed
%clean	N	Shell script	Cleans up the sources after a build is done
%pre	N	Shell script	Steps to take before installation of package
%post	N	Shell script	Steps to take after installation of package
%preun	N	Shell script	Steps to take before removal of package
%postun	N	Shell script	Steps to take after removal of package

Note that RPM makes heavy use of macros. The file **/usr/lib/rpm/macros** contains those already available and may be worth examining.

Details on RPM Spec File

The Header Section

The Header section is a list assignments, of the format:

```
Attribute: Value
```

An example showing the attributes usually used is provided below:

```
Summary: A great application!
Name: my_app
Version: 1.0
Release: 2
License: GPLv2
Group: Applications/Text
Source: ftp://ftp.myserver.com/pub/my_app/my_app-1.0.tgz
URL: https://www.myserver.com/my_app/index.html
Vendor: The Best Software Company
Packager: A genius <genius@myserver.com>
Patch0: my_app-1.0.patch0
Patch1: my_app-1.0.patch1
BuildRoot: /var/tmp/%{name}-buildroot
```

The **Name**, **Version**, and **Release** values are used to construct the names of the source and binary RPM files. Most other values are fairly self-explanatory, and can be chosen at will by the packager.

Note that the **License** field was called **Copyright** in earlier versions of RPM.

The description Section

The **description** section is completely free form. To see the description of an existing package, try **rpm -qi** package. An example description:

```
%description
This program enables the user to do everything that could
possibly be useful under Linux.
```

The prep Section

The **prep** section of the header must take the **tar** file of the source code, unpack it, and then apply any patch files needed. Often, this section is extremely short. An example **prep** section would be:

```
%prep
rm -rf $RPM_BUILD_DIR/my_app-1.0
tar zxvf $RPM_SOURCE_DIR/my_app-1.0.tar.gz
patch -p1 < my_app-1.0.patch1
patch -p2 < my_app-1.0.patch2
```

Modern versions of **rpmbuild** do not use such detailed **prep** sections. They can simply do:

```
%prep
%setup -q
%autosetup
```

The changelog Section

The **changelog** section lists a history of changes to the package. For example:

```
%changelog
* Tue Feb 29 2001 Mr Somebody <sbody@foo.com>
- important fix: cleaned up a bug that trashed the filesystem.

* Tue Feb 15 2001 Another Somebody <abody@foo.com>
- made minor changes to the initialization routine.
```

The build Section

The **build** section contains the commands needed to build your program. If your **prep** section did its work correctly, this should be one command:

```
%build
make
```

The install Section

The **install** section copies files to their final place in the filesystem. This is run after building the binaries, but not when the package is installed. When the binary is built, RPM will automatically package up the files that were put in place by the install section, then during installation, it will copy them back.

Usually, developers put an **install** target into makefiles that do this copying, which makes the install section of the **spec** file simple:

```
%install
make install
```

The files Section

The **files** section of the **spec** file lists all of the files that the **install** section copied into place. Files may be prefixed by an adjective that changes how RPM treats these files. The possible adjectives are:

Adjective	Description
%doc	Marks documentation from the source package to be included in a binary install. Multiple documents can be listed on the same line, or on separate lines. If no path for a documentation file is specified, then it will be put in a directory named **/usr/share/doc/package-version-build**.
%config	Mark configuration files. When you uninstall a package, any configuration files which have been changed will be saved with a **.rpmsave** extension.
%dir	By default, if you include a directory in the files section, the directory and all files it contains are put in as part of installation. Using **%dir** means that only the directory, and not the files that it contains, will be copied when the package is installed.
%attr	Set attributes for the file. An example would be **%attr(666,root,root)**, where the three fields are mode, owner, and group (just like the output of **ls -l**). Defaults are gotten by using a **-** in a place.
%defattr	The same **as %attr**, but you do not add a file to it. Instead, all files listed after the **%defattr** will automatically get the new attributes (unless they have their own **%attr** directive).

An example **files** section would be:

```
%files
%doc README
/usr/bin/my_app
%config /usr/bin/my_app_configure
/usr/man/man/my_app.1
%attr(600,-,root) /usr/lib/my_app/my_data
```

RPM Dependencies

In the **spec** file you may specify three types of dependency information:

- Capabilities that this package provides
- Capabilities that this package requires
- Packages that this package requires.

A capability is a required function or class of functions. You can see what libraries a package requires with:

```
$ rpm -qR package
```

as in:

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>rpm -qR gzip
/bin/sh
/bin/sh
/bin/sh
/sbin/install-info
coreutils
libc.so.6()(64bit)
libc.so.6(GLIBC_2.14)(64bit)
libc.so.6(GLIBC_2.17)(64bit)
libc.so.6(GLIBC_2.2.5)(64bit)
libc.so.6(GLIBC_2.3)(64bit)
libc.so.6(GLIBC_2.3.4)(64bit)
libc.so.6(GLIBC_2.4)(64bit)
libc.so.6(GLIBC_2.6)(64bit)
rpmlib(CompressedFileNames) <= 3.0.4-1
rpmlib(FileDigests) <= 4.6.0-1
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rtld(GNU_HASH)
rpmlib(PayloadIsXz) <= 5.2-1
c7:/tmp>
```

You can also see what libraries a package provides as in:

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>rpm -q --provides gzip
/bin/gunzip
/bin/gzip
/bin/zcat
bundled(gnulib)
gzip = 1.5-9.el7
gzip(x86-64) = 1.5-9.el7
c7:/tmp>
```

The names you see as the capabilities is not the full path name of the library; instead, it is what is called the **soname** of the library.

Any dynamic libraries in the files section of the **spec** are automatically added as capabilities that the package provides. In addition, RPM will automatically run scripts (called **find-requires** and **find-provides**) that determine which dynamic libraries your binary requires, so these will automatically be added to the list of capabilities that your package requires.

If your package has other requirements besides dynamic libraries, you can specify that another package must be installed by putting code of this format in the header section of your **spec** file:

```
requires: package
requires: package >= version
requires: package >= version-build
```

In addition to the **>= test** (which requires that a package of the version specified or later is installed), you can also use **>**, **=**, **<**, and **<=**.

Debian Package Creation Workflow

Building a Debian package requires many more files than does the RPM system, where almost all the complications go into the **spec** file.

The most authoritative documentation is the Debian New Maintainers' Guide.

At first glance, it is almost frighteningly complicated. However, there exist auxiliary utilities such as **debuild** and **cdbs** which can automate many tasks, provide templates for files which do not often have to be complex, and give command line interfaces for updating the various configuration files.

We will begin by explaining the basic package building workflow:

- First you will need the upstream package source, provided as a tarball in the form:

```
package-version.tar.gz
```

such as **someprogram-1.0.tar.gz**. Note the use of the hyphen (**-**) between the package name and the version number. The system is very picky about the name; it must contain only lower case letters, digits, plus and minus signs and periods. It cannot contain dashes or underscores.

- You will also need any patches that need to be made to the upstream source.
- The source will be expanded into **someprogram-1.0** and the original source tarball will be preserved in:

```
package_version.orig.tar.gz
```

Note the use of the underscore which has replaced the hyphen in the name; this can be very confusing!

- A directory **someprogram-1.0/debian** is created, and will be filled with all the files needed by the package building system.
- Source and binary packages are built from these ingredients; the binary package has the **.deb** suffix; we will talk about what a source package looks like later in an exercise (unlike RPM, where the source package is just one file, Debian source packages contain several files).