EE5012 UNIT 2 Objectives:

- To describe and explain: What is “Bash” and it’s significance to the UNIX/Linux OS
- To introduce some common shell commands and their usage
- To explore some of the main shell utilities such as “grep”
- To provide an overview of the “*/proc*” filesystem in UNIX/Linux

Shell Commands

1. Getting started

The shell

Before you start to write any programs under UNIX/Linux it is most important that you are confident with the basic shell commands. The word “**shell**” is a term for the user command level interface, where a user can directly write commands to instruct the computer to perform certain actions. The UNIX/Linux shell, sometimes referred to as a command interpreter, provides a standardised approach for interacting with the system. It is the most widely used utility program in Linux and offers the following features to the users:

i) **Interactive Environment:** The shell allows the user to create a dialogue between the user and the host system. This dialogue lasts until the user ends the session.

ii) **Shell scripts:** Shell script is the user interface to the computer system for the Linux operating system. The shell contains internal commands which can be utilized by the user. Shell scripts are groups of Linux commands strung together and executed as individual files.

iii) **Input/Output redirections:** Linux commands can be instructed to take their instructions from files, and not from the keyboard. The shell also allows the user to place the output of commands into a file and not on screen of the terminal. Output can also be redirected to other devices, such as to a printer or another terminal on the network.

iv) **Piping Mechanism:** Linux supplies ‘***pipe***‘ facility that allows the output of one command to be used as input in another Linux command. The Linux file system contains many utilities that are useful for such purpose.

v) **Metacharacter facility:** Apart from the normal characters found on a keyboard, (letters and digits) the shell provides the use of ‘***metacharacters***‘. These allows the user to apply selection criteria when accessing files, for example the user could access all files that begin with the letter ‘***m***‘.

vi) **Background processing:** The true multi-tasking facilities allow the user to run commands in the background. This allows the command to be processed in background while the user can proceed with other task in foreground. When background task is completed, the user is notified.

UNIX/Linux systems can support different shells. This document will introduce the Bash shell in particular. The word Bash is an acronym for 'Bourne-Again SHell' and it is a pun on Stephen Bourne's classic UNIX Bourne shell.

In summary, the Bash shell has been chosen for this document as it is a good general-purpose modern shell, and it is the default shell on many systems including Linux and Apple OS X. Bash has been ported to Microsoft Windows and is available with Cygwin, MinGW and DOS with the DJGPP project. Bash is available on Android using terminal emulation applications such as Terminal IDE and Termux.

2. Introducing Some Common Commands

Some of the more common UNIX shell commands will be introduced. The first command that we will look at is the **cat** (concatenate) command. Before we attempt to try this command, let's look at some ways on getting help with using a command. We will take the **cat** command as a first example command to see what help is available for this and other commands.

2.1 The "man" and "info" pages

The shell includes command manual pages, which are the so-called **man** pages. This is effectively an on-line user's manual for the shell commands. For each shell command you can call up a text description for the command by simply typing **man** *command name*. The description for each command is very complete but not always easy to understand by the novice. In addition to the man pages, there are information pages about each command, referred to as the **info** pages. These pages are accessed using the **info** command.

Type **man cat**

The screen will show an overwhelming amount of information about the **cat** command. You do not need to study all of this information now, rather use the **man** pages as reference information. A **pager** utility is used to show all of the man page content information on the terminal screen. Try the following keys to view the screen:

Hit the **space bar** to move down a screen

Hit the **b** key to move back up a screen

Hit the **up arrow** ↑ key to move up a line

Hit the **down arrow** ↓ key to move down a line

Hit the **q** key to exit the man pages

The **apropos** command is useful to search all the **man** pages and display a list of topics that are related to a query. The apropos command is effectively a wrapper for the *"man -k"* command.

2.2 The “info” and “help” pages

The **info** (information) pages are similar to the **man** pages.

Type **info cat**

Use the same pager keys as the **man** pages to navigate the **info** pages. Again, please note that these pages are available as reference information and do not need to be studied right now.

The help option

There is also a simple help option for most commands.

Type **cat - - help**

A summary display of the command options is displayed. Note the use of the double hyphen in calling the help option.

2.3 The command prompt

Traditionally, a shell prompt ends with one of the symbols $, % or #. The $ indicates a shell that is compatible with the Bourne shell, i.e. POSIX shell, Korn shell, or Bash shell. The % indicates a C shell, e.g. csh shell or tcsh shell. The # indicates the shell is running under a superuser account (root account).

So, the **$** symbol is the generic shell prompt for the Bash shell. For example the following command could be typed at the shell prompt:

$ whoami

The computer will reply as follows (assuming that your name is **john_smith**)

$ whoami
john_smith

As stated, the **#** symbol is used as the shell prompt when you are logged on as a **root** user. However, you should avoid using root user sessions, until you are an experienced user, so as to avoid any unnecessary accidental harm to the system.

2.4 The "cat" command

The **cat** (con**cat**enate) command is useful for displaying the contents of a text file and for concatenating files.

Example: Type **cat verse_1**
You will see the contents of the **verse_1** file listed to the screen

Example: Type **cat verse_1 verse_2 > big_file**
This command concatenates the two files and directs the output to a file called **big_file**.

Example: Type **cat verse_3 >> big_file**

Here the >> operator will cause the output of file **verse_3** to be appended to file **big_file**, i.e. big_file is not overwritten but rather the file verse_3 is added to big_file.

Now, create a file called **full_poem** using the **cat** command as follows:

Type **cat title verse_1 verse_2 verse_3 > full_poem**

Now type **cat full_poem** to view the contents of the file **full_poem**.

2.5 The "echo" command

The **echo** command is used, typically, to output text strings to the screen.

Try the following examples:

echo Hello outputs 'Hello' to the screen
echo "Hello" also outputs 'Hello' to the screen

The **-e** option is used to interpret backslashed escape characters in the string. Try the following examples:

echo -e "Hello \n" outputs 'Hello' to the screen and adds a new line

The **-n** option can be used to specify that the default new line is not to be generated.

There are many options for the **echo** command. Look these up in the **man** pages.

2.6 The "printf" command

It is interesting to note that the Bash shell also supports the **printf** command. The **printf** command does a formatted print based on the well-known C language **printf()** statement. It is intended to be a successor for the **echo** command and has a richer feature set. It is the preferred POSIX solution towards standardized code.

Here is a simple **printf** command example:

printf "Hello \n" outputs 'Hello' to the screen, adds a new line.

2.7 The “more” and “less” commands

For outputting large files to the screen the **cat** command is not very useful because we will see the entire file data rolling too quickly up the screen. The **more** command outputs just one screen full (e.g. 24 lines) of information at a time, under control of a pager. We already introduced the pager with the **man** command above.

Note, the **more** command is similar to the **less** command. Some systems support the **less** command but not **more**. However, l**ess** is not **more** – but **less** is similar to **more**, more or less! In fact **less** has more features than **more**. For example **less** supports backwards paging.

Hitting the **space bar** will output another screen page of data and so forth. Hit the **b** key to go back a page. A message at the bottom of the screen will tell you how much of the file has been displayed. Hit the **enter** key to display the next page. Hit the **q** key to quit.

Create a larger text file using the following commands:

Type **cat full_poem > large_file**

cat full_poem >> large_file

cat full_poem >> large_file

Display **large_file** to the screen, first using **cat** and then using **more.** You will appreciate the difference between these two commands.

2.8 The “pwd” command

The **pwd** (**p**rint the current **w**orking **d**irectory) command will tell you where you are within the directory structure.

Simply type **pwd**

The response will tell you where you are within the directory structure e.g.: **/home/hristo/EE5012**

2.9 The “ls” command

The **ls** command (**list**) is a very useful shell command, which can list files. The command has a wide range of options. Look at the **man** pages for the **ls** command (type **man ls**), or use the help facility by typing the **ls --help | less** command.

Simply type the command **ls** at the command prompt. You will see the list of file names sorted down columns in alphabetical order, something like the following:

```
hristo@hristo-lubuntu18:~$ ls
countDir  Downloads  longFile  opera-stable_50.0.2762.45_amd64.deb  usedSpace
countFile  ET4345  mem.txt  Pictures  Videos
Desktop  ET4725  Music  Public
busy_wait  Documents  freeDisk  mypipe  Templates
```

Now try the command: **ls -l** *(NB this is dash elle, not dash one)*

You will see a response something like this:

total 52468		**user**	**group**	**size**	**date**		**time**	**name**
-rw-rw-r--	1	hristo	hristo	127	Dec	6	09:53	**busy_wait**
-rw-rw-r--	1	hristo	hristo	3	Oct	23	13:24	**countDir**
drwx------	1	hristo	hristo	3	Oct	23	13:24	**countFile**
-rwxr--r--	2	hristo	hristo	4096	Oct	14	18:59	**Desktop**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Documents**
drwxr-xr-x	2	hristo	hristo	4096	Nov	8	15:05	**Downloads**
drwxrwxr-x	10	hristo	hristo	4096	Nov	9	11:12	**ET4345**
drwxrwxr-x	12	hristo	hristo	4096	Nov	8	12:23	**ET4725**
-rw-rw-r--	1	hristo	hristo	9	Oct	23	13:24	**freeDisk**
-rw-rw-r--	1	hristo	hristo	54	Oct	23	13:24	**longFile**
-rw-rw-r--	1	hristo	hristo	56994	Sep	7	12:41	**mem.txt**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Music**
prw-rw-r--	1	hristo	hristo	0	Dec	6	10:51	**mypipe**
drwxr-xr-x	2	hristo	hristo	4096	Jan	23	10:29	**Pictures**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Public**
drwxr-xr-	2	hristo	hristo	4096	Sep	5	13:34	**Videos**

The purpose of each field will be explained a little later on.

Try **ls -l /** to list the root directory

Try **ls -d */** to list directory entries only (a useful command!)

2.10 The Glob Constructs (wild cards)

The so-called wild cards or file globs can be used in specifying a file name, using filename patterns that include special characters such as ***** and **?** etc. Under UNIX/Linux these are referred to as **glob constructs**. This type of pattern matching is part of a subject called *regular expressions*, which can support powerful pattern languages, as will be introduced later on.

2.10.1 The ? (match single) Glob Construct

Any single character in a file name is made wild.

e.g.: **ls ?est_demo** might list files such as the following (fictitious example) :

pest_demo best_demo test_demo

2.10.2 The * (match any) Glob Construct

Any string of zero or more characters is wild

e.g.: **ls *t_demo** might list the following:

pest_demo best_demo test_demo bat_demo bert_demo t_demo

2.10.3 The [ ] (match list) Glob Construct

Any character inside the square brackets can be used to match **one** (and only one) character in the file name.

e.g.: **ls num[xyz]test** might list the following:

numxtest numytest numztest

Since only one character within the bracket can be selected, a file called **numxytest** would not be selected.

ls [a-e]test will list any file name whose first character in the name is in the range **a** to **e** and the rest of the file name is 'test'. For example, a valid list might be:

atest btest ctest dtest etest

Table 1. Shell Metacharacters

Symbol	Meaning and Usage
<	Redirection: Get input for the command to the left from the file listed to the right of this symbol. Example: `sort < filename.txt # This will print out the contents of filename.txt with the lines sorted.`
>	Redirection: Send the output of the command on the left into the file named on the right of this symbol. If the file does not exist, create it. If it does exist, overwrite it (erase everything in it and put this output as the new content). Example: `cal > filename.txt # This puts a current calendar into filename.txt`
>>	Redirection: Send the output of the command on the left to the end of the file named on the right of this symbol. If the file does not exist, it will be created. If it does exist, it will be appended. Example: `date >> filename.txt # This adds the current date and time to the end of filename.txt`
|	Pipe: This sends the output of the command to the left as the input to the command on the right of the symbol. Example: `cat filename.txt | grep it # This will print the lines in filename.txt that contain the string "it"`

$	Denotes that a string is a variable name. Not used when assigning a variable. Example:
	`prompt: my_variable="this string" # Assigning the variable`
	`prompt: echo $my_variable # Referring to the contents of the variable`
	Also used in parameter substitution to check if a variable is defined and determine a value. Example:

	`echo ${my_variable-other} # Print the value of my_variable if defined, otherwise print "other"`

\	Escape. Ignore the shell's special meaning for the character after this symbol. Treat it as though it is just an ordinary character. Example:
	`echo "I have \$300.00" # Print the $ instead of using it to look up a variable name.`
	Also used to put commands on mulitple lines. Example:
	`prompt1: cat filename.txt \ # Don't run yet...`
	`prompt2: date \ # Still don't do anything...`
	`prompt2: cal # No backslash on the last one, so run all the command.`

( )	Grouping commands. If you want to run two commands and send their output to the same place, put them in a group together. Example: `(cal; date) > filename.txt # Put a calendar and the date in filename.txt`

{ }	Used in special cases for variables with the $. One use is in parameter substitution (see the $ definition, above), the other is in arrays. Example:
	`numbers=( 1 2 3 4 5 )`
	`echo ${numbers[1]}`

“	Quoting. Used to group strings that contain spaces and other special characters. Example: `my_variable="This is a test." # Assign a string to the variable`
‘	Quoting. Used to prevent the shell from interpreting special characters within the quoted string. Example: `my_variable='This is a backslash: \' # Assign the string to the variable`
`	Unquoting. Used within a quoted string to force the shell to interpret and run the command between the backticks. Example: `my_variable="This is the date: `date`" # Store the string AND the output of the date command`
&&	Run the command to the right of the double-ampersand ONLY IF the command on the left succeeded in running. Example: `mkdir stuff && echo "Made the directory" # Print a message on success of the “mkdir” command`
||	Run the command on the right of the double pipe ONLY IF the command on the left failed. Example: `mkdir stuff || echo "mkdir failed!" # Print a message on failure of the “mkdir” command`

&	Run the process in the background, allowing you to continue your work on the command line. Example:
	`john /etc/passwd & # Try to crack the passwords - this takes a couple hours, so do it in the background.`
	Also used in redirection when copying one stream into the same location as another stream. Example:
	`cat filename.txt > file2.txt 2>&1 # Send Standard Error (2) to the file2.txt where Standard Output (1) is going`

;	Allows you to list multiple commands on a single line, separated by this character. Example: `date;john passwd; date # Print the date, crack the passwords and print the date again afterwards - cheap benchmarking`
=	Assignment. Set the variable named on the left to the value presented on the right. Example: `my_variable="Hello World!" # Note that there is NO SPACE between the variable name and the string.`

2.11 Commands to Copy (cp), Move (mv) and Remove (rm) Files

2.11.1 The "cp" command

The **cp** command is the UNIX/Linux **c**o**p**y command.

For example, type: **cp verse_1 verse_temp**

Type **ls** to see that the new file **verse_temp** does now exist, and that it is a copy of the **verse_1** file.

2.11.2 The "mv" command

The **mv** command moves (**m**o**v**es) a file. It effectively renames the file.

For example, type: **mv verse_temp verse_demo**

Use **ls** to see what happened to the file. You will see that the file **verse_temp** no longer exists as it has been effectively renamed to **verse_demo**.

2.11.3 The "rm" command

The **rm** command removes (re**m**oves) a file, i.e. deletes a file.

For example, type: **rm verse_demo**

Use **ls** to satisfy yourself that the file **verse_demo** no longer exists.

2.12 Using Directories

Earlier we used the **pwd** command to see where we were in the directory structure. Now we will move about within the directory structure and create a new directory. The **cd**, **mkdir** and **rmdir** commands will be shown to respectively: change, make and remove directories.

2.12.1 The "ls" command

For practice try these commands:

ls . lists the files in your current directory

ls .. lists the files in your parent directory

ls / lists the files in the root directory

ls *pathname* lists the directory specified in the pathname

Some more **ls** options:

ls -R lists files and sub directories (recursively)

ls -a lists **a**ll files including files which begin with '**.**' i.e. the UNIX utilisation files

ls -l list file details, see later for more detail (this option is **dash elle**).

ls –1 lists files in a single column (this option is **dash one (-1)**, not **dash elle (-l)**)

Examples to list information on directories:

ls */ lists the contents of all the subdirectories
ls –d */ lists only the directories in the current directory

Use the **man** pages to learn more about the **ls** command and its options.

2.12.2 The "cd" command

The **cd** command (**c**hange **d**irectory) changes the current directory. Note, when you log onto a UNIX/Linux system, your current directory is, by default, your **home** directory e.g. **/home/hristo**.

For practice try the following examples to change to other directories, each time you change to a new directory you can verify by using **pwd** to show where you are and use **ls** to see the contents of the new directory:

cd .

cd ..

cd /

cd .. /../

cd *pathname* (try some specific path name)

Make sure you understand what each one of the above commands do.

If you get lost somewhere within the maze of the directory structure and want to go back to your home directory, simply type:

cd ~ changes to your home directory (tilde (~) represents your home directory).

or, even more simply, type:

cd

2.12.3 The "mkdir" command

The ***make directory*** command, **mkdir**, is used to create a new directory.

For example, make a new directory called **my_test** and copy all the files from your home directory to the **my_test** directory. Try the following steps:

cd ~	go to your home directory
mkdir my_test	make a new directory called **my_test**
cd my_test	change into **my_test** directory
ls	this will show that no files exist here yet as this is a new directory
cp ../* .	copies all files from parent directory to **my_test** directory

Obviously, you could have copied these files without actually going into the new directory, but there are many ways to navigate the file system.

If you have directory files in the parent directory you will receive an error message something like:

cp: ../my_test: omitting directory

To copy directory files you must specify the **-r** option (see manual pages for **cp**)

Type **cp -r ../* .** Now this command should copy all files including directory files. Try this to ensure that it works.

2.12.4 The "rmdir" command

The **rmdir** (**rem**ove **dir**ectory) command will remove the specified directory or directories.

Switch back to your home directory (**cd ~**) and try to remove the **my_test** directory by typing:

rmdir my_test

Just as you would have expected, you get a message telling you that the directory is not empty, and thus removal of the directory is denied.

How would you remove a directory and its contents without first deleting the contents?

You would use: **rm -r my_test** Try this, it works!

But this is not the **rmdir** command, why does it use the **rm** command? Well, read the **rm** man pages and you will get some explanation.

2.13 File Details

2.13.1 Exploring files using the “ls” command

Type **ls -l** and you will get a screen something like the following:

total 52468		**user**	**group**	**size**	**date**		**time**	**name**
-rw-rw-r--	1	hristo	hristo	127	Dec	6	09:53	**busy_wait**
-rw-rw-r--	1	hristo	hristo	3	Oct	23	13:24	**countDir**
drwx------	1	hristo	hristo	3	Oct	23	13:24	**countFile**
-rwxr--r--	2	hristo	hristo	4096	Oct	14	18:59	**Desktop**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Documents**
drwxr-xr-x	2	hristo	hristo	4096	Nov	8	15:05	**Downloads**
drwxrwxr-x	10	hristo	hristo	4096	Nov	9	11:12	**ET4345**
drwxrwxr-x	12	hristo	hristo	4096	Nov	8	12:23	**ET4725**
-rw-rw-r--	1	hristo	hristo	9	Oct	23	13:24	**freeDisk**
-rw-rw-r--	1	hristo	hristo	54	Oct	23	13:24	**longFile**
-rw-rw-r--	1	hristo	hristo	56994	Sep	7	12:41	**mem.txt**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Music**
prw-rw-r--	1	hristo	hristo	0	Dec	6	10:51	**mypipe**
drwxr-xr-x	2	hristo	hristo	4096	Jan	23	10:29	**Pictures**
drwxr-xr-x	2	hristo	hristo	4096	Sep	5	13:34	**Public**
drwxr-xr-	2	hristo	hristo	4096	Sep	5	13:34	**Videos**

What does each field in the above represent? First of all, each UNIX/Linux user is assigned a unique user ID number. When you create a file, you normally become the file **owner**. The owner of the file can specify access permissions to the file, e.g. **read** access permission, **write** access permission, **and execute** access permission.

rwx access permissions for regular files:

r can **r**ead a file
w can **w**rite to a file
x can e**x**ecute a file

rwx access permissions for directory files are as follows:

r can **r**ead names of entries in the directory
w can create new files and sub-directories
x can check to see if a file or sub-directory e**x**ists by specifying the file or sub-directory name. Not as strong as the **r** permission.

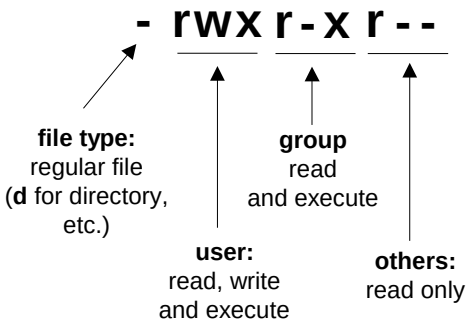
UNIX/Linux also has the **group** concept. A number of users can belong to a group. In fact one user can belong to several groups but one group will be the user's primary group. Just like the owner’s access permissions, the group will also have access permissions, i.e. another user in the group can read or write or execute your file. The user access permissions are noted using the **rwx** access convention also.

So, for a given file the **user** and the **group** access permissions are defined; but what file access permissions should apply to everyone else? Well, there are another three **rwx** bits to define the access permissions for everyone else. Everyone else is often referred to as '**others**'.

Let's look at the following line which results from a **ls -l** command:

-rwxr-wr-- 2 hristo vboxsf 512 oct17 15:11 test_file

The first field is used to define the file type and the access permissions, as follows:



The second field (showing the number **2** in this example) shows that there are two **links** to the file (the concept of a link will be explained later on.)

The next field tells us who is the file's owner (**hristo** is the owner in this example**)**

Next field (**vboxsf** in this example) specifies the file's group (ignore for now)

The next field shows the file size in bytes (**512** bytes in this example)

The next field shows the **date** and **time** stamps for the file (**oct 17 15:11 in this example**)

The final field shows the file name (**test_file** in this example)

Note the access permissions are sometimes referred as the '9-bit' permission field. In the above example the **rwx, r-x, r- -** code is represented internally in the UNIX file system by the binary code **111,101,100** (or 754 on octal) where each bit position represents a permission status.

2.14 Touching files

2.14.1 The "touch" command

The touch command is useful. It updates the last *modification time* and *access time* of named files to the current system time. If a specified file does not exist, then **touch** will create that file with a zero size.

Try the following:

touch BrandNew

Use the **ls -l** command to verify that a new file called **BrandNew** has been created and that this file has a zero size.

2.15 Changing File Owner And Access Permissions

2.15.1 The "chmod" command

The **chmod** command will allow you to change a file's access permissions. The command uses the following parameters:

u means **user** who owns the file
g means **group** that owns the file
o means **others**
a means **all**

\+ means permission **on**
\- means permission **off**
= means add permission (on) but remove all other permissions

r means **read** permission
w means **write** permission
x means **execute** permission

For example:

chmod a-w test_file	says you want to **turn off** the **write** permission for **all** i.e. user, group and others.
chmod u+rwx test_file	says you want to **turn on** the **read**, **write** and **execute** permissions for the **user**.
chmod go+w test_file	says you want to **turn on** the **write** permission for the **group** and **others**.
chmod 644 test_file	this sets the file's permissions to (rw - r- - r- -); saying to give **read/write** permission to the **user** and **read-only** permission to everyone else.

For a given file, practice revoking (turn-off) and granting (turn-on) access permissions (rights) for various files and directory files. To be on the safe side make some temporary files to practice on.

2.15.2 The "chown" command

The **chown** (**ch**ange **own**er) command changes the file ownership. The user level access here is restricted. Look up this command in the **man** pages to find out more.

The **chown** command allows a root user (**chown** is not available to ordinary users) to change the ownership for a file.

For example:

chown claire test_file	changes the owner of the file **test_file** to **claire**.
chown daniel test_file	changes the owner of the file **test_file** to **daniel**.

2.16 Finding Files

2.16.1 The "locate" command

The **locate** command will look for a file which matches a search file name. The **locate** command will quickly return any matching results, as **locate** searches a procreated database file. However, this database file needs to be updated frequently so that it can contain an up-to-date list of current files. The **updatedb** command will perform the update. Note that root user privileges are required to run **updatedb**.

Note, the **slocate** command, secure locate, is a better command as it does not search directories where the user does not have access privileges.

Try running the **updatedb** command on your system, if you have the rights to do so. Beware that the command may take a long time to complete.

Try using the **locate** command as follows:

locate locatedb

The results will advise if a **locatedb** file or an **slocatedb** file exists, and will show the path to the files.

2.16.2 The "find" command

Whereas the **locate** command searches a database file for matching files, the **find** command will actually parse through the real file system, searching for files which match user defined criteria. Hence the **find** command will be much slower than the **locate** command, but it acts on current file information and provides great flexibility in defining the search criteria.

WARNING – it might seem that a find command will take forever to complete!

The **find** command will perform a recursive search through the directory structure.

The general format for the **find** command is:

find *pathname* *expression* *-print*

Note the **–print** option is enabled by default so there is no need to specify it when you use **find**. The **find** command is recursive in its search through directories.

Read the **find** command's man pages.

Examples: Try the following **find** commands and satisfy yourself that you understand what each command is doing:

find . -name "verse*"
Finds all files and directories, starting at the current directory level (.) which have the string 'verse' as the start of the file or directory name, and lists the output to the screen.

find / –user Ann2017
Finds all files and directories starting at the root directory level (/) which are owned by the user Ann2014.

find . -type f
Finds all normal files (f), starting at the current directory level (.) and lists the output to the screen.

find . -ctime 7
Finds all files starting at the current directory level (.) which have been changed seven days ago (-ctime 7) and lists the output to the screen.

find . -ctime -8
Finds all files starting at the current directory level (.) which have been changed within the past seven days (-ctime -8) and lists the output to the screen.

find . -atime 3
Finds all files starting at the current directory level (.) which have been accessed three days ago (-atime 3) and lists the output to the screen

find . ! -atime -3
Finds all files starting at the current directory level (.) which have **not** been accessed within the past two days (! -atime -3) and lists the output to the screen.

find / -type d
Finds all directory files (i.e. files of type d), starting at the root (/) directory, and lists the output to
the screen.

There are many other interesting search options for the find command. Look them up!

2.17 Redirection

2.17.1 Redirection (>, <, >>)

By default the output from a command is written to the screen, and a command's input is assumed to be from the keyboard. There is also the concept of error messages being displayed to the screen.

The redirection symbols can be used to specify files as inputs or outputs for a command, as shown in some of the following examples.

Redirect a command's output to a file using the > character

The command **cat testfile** will print the content of **testfile** to the screen by default.

The command **cat testfile > fileOne** will print the content of **testfile** to the file named **fileOne**. Note, if **fileOne** had already existed, then its contents would have been overwritten.

Append a command's output to a file using the >> characters

The command **cat testfile >> fileTwo** will print the content of **testfile** to the file named **fileTwo**. However, the file **fileTwo** will not be overwritten, but rather the **testfile** content will be appended to the file **fileTwo**.

You can specify redirection in command lines.

What will the following command do?

ls -l > logfile

And in what way is this command different?

ls -l >> logfile

A file becomes an input for a command by using the < character

For example, the **sort** command is useful to do a line by line sort on an input file (the sort command will be explained shortly).

Try this example:

sort < somefile the **somefile** file is sorted by line and outputted to the screen

2.17.2 An interesting use of redirection using the “cat” command

Often we use the **cat** command as follows: **cat file_name**, which will list the contents of **file_name** to the screen. The **cat** command is taking **file_name** as its input and it is outputting to the standard output device, i.e. the console screen. The command **cat > scrapfile** specifies the file **scrapfile** as its output and the input, which is not specified in this command, defaults to the console i.e. the keyboard.

Type **cat > scrapfile**. Now type some characters on the keyboard and they will be written to the file
scrapfile. To terminate the input type the end-of-file control: CRTL+D.

This is a very quick way to generate a small text file without using an editor.

Try this example:

Type: **cat > hamlet**

Now the **cat** command is expecting characters from the keyboard.

Type the following text:

To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them?

Hit the CRTL+D keys to terminate the input text.

Now type:

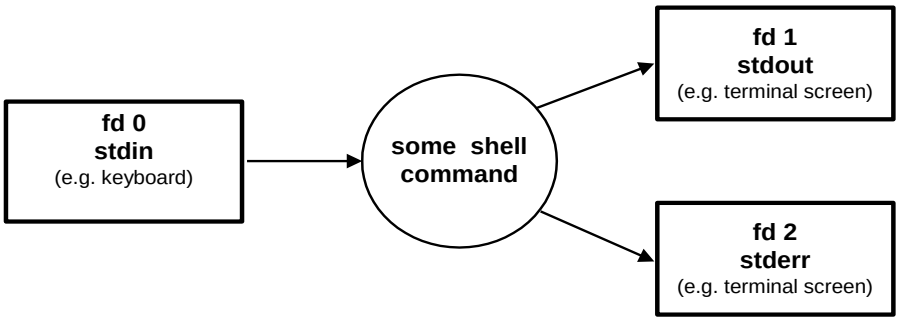
cat hamlet Now you know how to create a very simple text file without using an editor.

3. Input / Output (I/O)

The shell commands use file descriptors (referred to as ‘fd’) to identify the standard input (**stdin**), standard output (**stdout**) and standard error (**stderr**), as follows:

Standard I/O	**File descriptor**	**Default device**
stdin (standard input)	0	keyboard
stdout (standard output)	1	screen
stderr (standard error)	2	screen

A file descriptor is simply a number that is assigned to a device, or a file (or other such as pipes, sockets, or terminals), for convenient internal reference by the operating system kernel. By default **fd 0** is assigned to the standard input (**stdin**), **fd 1** is assigned to the standard output (**stdout**) and **fd 2** is assigned to the standard error (**stderr**).



Some examples of the special redirection operators are given below. We have already looked briefly at the shell’s basic redirection operators for output redirection (>), input redirection (<), and appending (>>). Now, we will look at some commonly used redirection operators that use the file descriptors. We can direct a file descriptor to a specified file using the following operator, where **n** is the file descriptor number:

n > filename

A common example is to redirect the standard error (the number for the stderr fd is 2) to a file using the code:

2 > filename

Assume that your current working directory contains four files which have file names starting with the letter ‘d’ and there are no file names that start with the letter ‘z’. Assume that you will type the following command, to list all files that begin with the letter ‘d’ and all files that begin with the letter ‘z’.

ls d* z*

Assume the command returns the following information:

ls: z*: No such file or directory

dan.tar document1 document2 duplicatefile

The first line in the output is an error message which is intended for the **stderr** device and the second line is a normal output for the **stdout** device. Since the screen is the default output for both **stdout** and **stderr**, both messages appear on the screen, as seen above.

Assume that you type the following command:

ls d* z* 2 > errorfile

This command causes the **ls** command to send its **stderr** output (remember **stderr** has the file descriptor number 2, hence **2 >)** to a file called **errorfile**. Now only the **stdout** output is sent to the screen. Thus the following output appears on the screen:

dan.tar document1 document2 duplicatefile

The **2 >** character pair is saying that the file descriptor number 2 (i.e. **stderr**) is redirected to the **errorfile** file. If the character set **2 >>** was used, then the **stderr** output would have been **appended** to the **errorfile** file.

3.1 The &n operator

The **&n** represents the file descriptor **n**, so **>&n** is saying that the standard output of a command is to be redirected to the file descriptor **n**. So we use the **>&** to duplicate **fd 1** and put this duplicate in fd **n**.

Here is an example:

ls d* z* 2 > errorfile1 >&2

Now, both the **stderr** and **stdout** are directed to the file **errorfile1**. The command is evaluated from left to right (as usual). The **2 > errorfile1** code causes the **stderr** (file descriptor 2) to be redirected to the file **errorfile1**. The **>&2** code causes the **stdout** output to

be redirected to the **stderr** (file descriptor 2) device, and since the **stderr** has been already redirected to the file **errorfile1**, in this command, then **stdout** is directed to **errorfile1**.

Some additional comments

1 >&n has the same meaning as **>&n** above, since the **stdout** (file descriptor **1**) was assumed in **>&n**. Thus the command line above could have been written as:

ls d* z* 2> errorfile1 1 >&2

The **m > &n** character set shows a general representation, where the file descriptor **m** is redirected to the file descriptor **n**.

The **<&n** character set shows input redirection where the file descriptor **n** is redirected to the standard input.

4. Sorting Text Files

4.1 The sort command

The sort command will sort the contents of a text file based on some well-defined criteria.

For example, if you use the **cat** command to display the content of a file called McGee:

cat McGee

There are strange things done 'neath the midnight sun
By the men who moil for gold.
The arctic trails have their secret tales
That would make your blood run cold

Now, you can sort the **McGee** file's output as follows:

sort McGee

By the men who moil for gold.
That would make your blood run cold
The arctic trails have their secret tales
There are strange things done 'neath the midnight sun

Now try the **sort** command on some text file of your own, e.g.: **sort full_poem**

You will see that the **sort** command sorts the file to the screen display, by line, in alphabetical order. The content of the input file is unchanged. To sort the input file to an output file try the following:

sort full_poem > poem_sorted

Use the command **cat poem_sorted** to convince yourself that the sort did happen.

Multiple files can be sorted. For example try the following command and then inspect the output file:

sort verse_1 verse_2 verse_3 > sort_file

The **sort** command includes some powerful sort options. Look up the **sort** command in the **man** pages. There are interesting options. For example, look at the following command that uses sort's **-g** and **-r** and **-k** options:

As an example, type **ls -l > myList** to create a file called **myList** that contains a listing of your files, in the following format:

-rw-r--r-- 1 hristo myGroup 448 Jan 16 09:03 dec1
-rwxr-xr-x 1 hristo myGroup 5257 Jan 16 09:03 delay
-rwxr-xr-x 1 hristo myGroup 5266 Jan 16 09:03 delay_prog
etc…

This creates a file called **myList** which contains the list of files in your current directory.

Now, assume you want to sort these files in order of the size of the actual files. Note, the 5th column contains the file size information, as seen in the listing above.

Type **sort -g -r -k5 myList**

This sorts the file, where **-g** specifies a general numeric sort, **-r** specifies a reverse order sort, i.e. the high numbers to the top, and **-k5** specifies that field 5 (column 5) is to be the key column for the sort.

Type **cat myList** (or **less myList** if you have a long file) to see that the file is actually sorted as required, i.e. in numerical order based on file size.

Of course we did not really have to use the **sort** command at all in the above example. We could simply have used the sort option in the **ls** command (ls -l -S). However, we did want to provide a simple example of how to use the **sort** command. In UNIX/Linux there is usually many different ways of achieving the same result.

5. Using Pipes

The use of pipes in commands allows the output from one command to become the input of another command. The | symbol denotes a pipe.

Try the following examples:

Create a file called **demo_file1** as a text file with a number of lines.

cat demo_file1 | more	The **demo_file1** text is sent to the **more** utility for screen display.
cat demo_file1 | sort | more	The **demo_file1** text is sent to the **sort** utility, where the lines are sorted and the output from **sort** is then sent to **more** for screen display.

5.1 The tee command with pipes

The **tee** command allows the splitting of a pipe so that a command's output can be simultaneously directed to the standard output, which is usually the next stage of the pipe, and to a file (or a named pipe).

An example usage of the **tee** command:

cat demo_file1 | sort | tee demo_file1_sorted | more

Here the **demo_file1** file is listed to the screen, using the **more** command. However, the sorted output is also sent to the **demo_file1_sorted** file using the **tee** command.

6. Search for Patterns with "grep"

In this section we will look at using regular expressions, the regular expression metacharacters, and searching for patterns. The **grep** utility will be introduced for these purposes.

Note, the GNU grep is used in the examples.

grep [Global Regular Expression Print]

The **grep** utility scans text files, looking for a string match.

The general format for the command is: **grep *string file***

Example:

Type **cd ~**	takes you to your home directory:

Type **grep the verse_1** displays the lines in **verse_1** which contain the string ‘the’

The **grep** command displays all the lines which contain the matching string 'the'. Optionally we could have used quotation marks, which is better in practice, as follows:

grep ‘the’ verse_1

The **grep** command is **case sensitive**, it knows the differences between uppercase and lowercase characters. Try the following two commands and note the different results:

grep ‘and’ verse_2

grep ‘And’ verse_2

To ignore the **case** of letters you should use the **-i** option. Try the following command and note the result:

grep -i ‘and’ verse_2

Note, **grep** searches for the expression one line at a time, it does not look for the expression across line boundaries.

The **–v** option can be used to find the lines where the specified string does **not** appear in the file. Try the following example, to display all the lines that **do not** contain the string ‘the’:

grep –v ‘the’ verse_2

6.1 Using “egrep”

The **grep** command has an Extended Regular Expression (ERE) mode that can be called using grep –E, i.e. by using the –E switch the extended features are used so that the expression is evaluated as an ERE as opposed to grep’s normal pattern matching. The **egrep** command was developed to provide a command to use **grep** in the extended mode. Here the following two commands are equivalent:

grep –E
egrep

In many cases we would simple use **egrep** to be inclusive of the extended cases.

6.2 Line and word anchors

6.2.1 The Line anchors

The **grep** examples above search for a simple string. A regular expression describes a pattern of characters. For example, the Metacharacter caret (^) represents the beginning of a line.

e.g.: **grep -i '^and' verse_2**

Here, within the quotation marks is a regular expression which says that we are searching for the string "and" (case insensitive because of the **–i** option) and this string must exist at the beginning of a line, as defined by the use of the ^ character.

As another example, type the following commands, which will list all of the **directory files** in your current directory to the file **temp_file**.

First, make a few directory files, using the **mkdir** command, if such directory files do not already exist.

ls -l > temp_file **temp_file** now contains the directory listing

grep '^d' temp_file now, all lines which start with the letter 'd' (i.e. directory files) are listed to **temp_file**.

Remember a directory file will have a 'd' character at the beginning of the line, such as in the following example where **donDir** is a directory file.

drwxr-xr-x 2 donal myGroup 4096 Jan 16 09:02 donDir

The metacharacter (**$**) represents the **end of a line**.

For example: Type **grep 'floor$' verse_2** and see the result, where only lines that finish with the string 'floor' are outputted.

6.2.2 The Word anchors

The \< and \> are the word anchors, where \< matches the start of a word and \> matches the end of a word. Consider the following two-line file called play.

$ cat play
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we can check lines that can match the pattern 'begin'. We will use a pipe in this example:

$ cat play | grep 'begin'
In the beginning we all sat down,
then Joe said we should begin to read aloud.

We used a pipe in the above example as this is a common way to use **grep**, but we would expect the same result from this command:

$ grep 'begin' play
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we check lines that can match the pattern ‘begin’ that must be at the start of a word.

$ cat play | grep '\<begin'
In the beginning we all sat down,
then Joe said we should begin to read aloud.

Now we check lines that can match the pattern ‘begin’ that must be at the end of a word.

$ cat play | grep 'begin\>'
then Joe said we should begin to read aloud.

We can use the following code to ensure that the pattern ‘begin’ is indeed as separate word, i.e. it is separated by spaces:

$ cat play | grep '\<begin\>'
then Joe said we should begin to read aloud.

However, if we want to find a string that is a separate word, it is easier to use the **-w**, as follows:
$ cat play | grep -w 'begin'
then Joe said we should begin to read aloud.
So, the **–w** is used to force a pattern to match only whole words. If we wanted to force a pattern to match only whole lines we would use **–x**.

6.2.3 Wildcards

In **grep** the Metacharacter **‘.** ‘ represents a single wild character that matches any one character.

For example: Type **grep -i ‘.he’ verse_3**

$ grep -i '.he' verse_3
I will find out where she has gone,
And kiss her lips and take her hands;
The silver apples of the moon,
The golden apples of the sun.

Note the matches such as 'The', 'she' and ' he' will be found. Notice how the 'he' as part of 'her', matched as ‘space he’.

Note, to display lines containing the literal dot character, use grep’s ‘-F’ option.

To find all the lines that contain a four letter word at the beginning of a line we could use:

```
$ grep '^.... ' verse_2
```
When I had laid it on the floor
With apple blossom in her hair

Note the space in the pattern '^.... ' in the above.

To find all the lines that contain a three letter word that begin with A or T at the beginning of a line we could use:

```
$ grep '^[AT].. ' verse_3
```
And kiss her lips and take her hands;
And walk among long dappled grass,
And pluck till time and times are done
The silver apples of the moon,
The golden apples of the sun.

Note the pattern '^[^AT].. ' would modify the above example to mean find the lines that do not start with A or T.

Both of the following statements are equivalent:

```
$ grep '^[AT].. '  verse_3’
$ egrep '^[AT].{2} '  verse_3 # note we used egrep here to support the quantifier.
```

Here the {} is a quantifier that determines how many times the previous character should occur.

The asterisk ‘*’ is used for matching multiple characters, the match is made on repetitions of a character. Say we want to match all words that start with ‘c’ and end with ‘t’, and we try the following:

```
$ cat verse_1 | grep 'c.*t'
```
And cut and peeled a hazel wand,
And moth-like stars were flickering out,
And caught a little silver trout.

Did we really want to accept the string: ‘fli**c**kering out’? Ok, we got what we asked for, but maybe not what we hoped for; we should have been more careful to specify word boundaries, so now we try the following:

```
$ cat verse_1 | grep -w 'c.*t'
```
And cut and peeled a hazel wand,
And caught a little silver trout.

In the above example note the use of ‘.*’ in **grep**. where the ‘.’ operator is followed by the ‘*’ operator. The ‘*’ operator specifies that the preceding item will be matched zero or more times, and in the example the ‘.’ is the preceding operator which specifies a match on any

single character. So, any expression consisting of a character followed by a '*' matches any number of repetitions of that character (and this could possibly be zero repetitions).

A summary, not a complete list in **grep**, of the regular expression operators is provided in the table below.

Operator	Interpretation
.	Matches any one characters
*	Matches zero or more of the previous characters
.*	Matches any number or type of characters
[ ]	Matches characters listed in the brackets
[^]	Does not match any characters that are listed in the brackets
-	Represents the range if it's not first or last in a list or the ending point of a range in a list.
^	Denotes the beginning of a line
$	Denotes the end of a line
\<	The empty string at the beginning of word is matched.
\>	The empty string at the end of word is matched.

As an example, consider the following code which will search for any word that begins with an uppercase letter in the range **B** to **D**, or a lowercase letter in the range **k** to **n**.

```
$ grep '\<[B-Dk-n]' verse_1
```
Because a fire was in my head,
And when white moths were on the wing,
And moth-like stars were flickering out,
And caught a little silver trout.

7. Finding Differences

7.1 The "diff" command

The **diff** command will compare two text files and display the differences.

Try the following steps:

a) Type **cp verse_1 verse_temp** makes an exact copy of **verse_1** called **verse_temp**

b) Using the editor modify the **verse_temp** file: change the word 'berry' to 'worm' and the word 'trout' to 'salmon'.

Now type the command **diff verse_temp verse_1**

The command output shows the lines which contain the differences, the (>) character signifies that the line is from the first file and the (<) character signifies that the line is from

the second file. You will see cryptic notes on the output such as (**4c4**) and (**8c8**). These are to advise what must be done to eliminate the differences. For example (**4c4**) means that line 4 of **verse_temp** must be changed to line 4 of **verse_1**, and (**8c8**) gives similar information about line 8. There are other messages which can be outputted by **diff** (e.g. **a** for append, **d** for delete etc.) but these can be discovered later.

8. Count words, lines and characters

8.1 The "wc" command

The **wc** command will tell you the number of lines, the number of words and the number of characters in a text file.

For example: Type **wc full_poem**

The command prints a message like:

35 172 917 full_poem

This is saying that there are 35 lines, 172 words and 917 characters in the file **full_poem**.

The **wc** command if often used to simply count the number of lines in a file, as follows:

wc -l myFile *(NB the dash elle here means count lines)*

Try this on a few sample text files.

9. Calendar

9.1 The "cal" command

The **cal** command displays a simple calendar. Try the following commands:

cal 7 2017

This outputs a calendar for July 2017; you might get the following result:

```
$ cal 7 2017
     July 2017
Su Mo Tu We Th Fr Sa
                 1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

10. Time and Date

10.1 The “date” command

The **date** command can set or display the system's current time and date. Try using the **date** command as follows:

hristo@hristo-lubuntu18:~$ date
Tue Feb 5 14:12:12 GMT 2019

In Bash, many of the commands are very flexible and contain more features and options than what might be first perceived. This is why it is always worth examining the range of options for each command. Consider the following example where we want to be very specific regarding what format we want to display for the date. The **%b** parameter represents the short name of the current month, **%d** represents the day of the month, and **%Y** is a four-digit representation of the year:

hristo@hristo-lubuntu18:~$ date "+%b %d, %Y"
Feb 05, 2019

10.1.1 Using the date command to measure UNIX’s lapsed time

The following command will give us the current time.

$ date +%s
1435845758

This is time in the UNIX’s style, where time is the number of seconds since the ‘UNIX epoch’, sometimes referred to as the POSIX epoch, which is the time 00:00:00 UTC on January 1, 1970.

Here is an interesting example, if non-intuitive, it is a niche use for the date command:

Type the following:

$ date +%N
538537300

Repeat the above a few times and look at the results, which appear to be random numbers. The **+%N** is an output option which shows the **nanosecond** portion of the current time. Since nanoseconds represent such a high time resolution, by the random use of this command we can generate informal random numbers.

11. Disk Storage Utilities

11.1 The “df” command

The **df** (**d**isk **f**ree) command displays information about space usage on the file systems. Use the **man** pages to help you to interpret the results and to see what options are available.

Type **df** to see information on the various mounted file systems.

Type **df .** to show information on the file system for the current directory. A response similar to the following will be seen.

```
hristo@hristo-lubuntu18:~$ df .
Filesystem     1K-blocks    Used Available Use% Mounted on
/dev/sda1      20315348 9773408   9486916  51% /
```

The report shows the file systems (i.e. disk partitions/volumes) sizes in **1kByte** blocks, with column 2 showing the full size, column 3 showing the **Used** space and column 4 showing the **Available** space. Column 5 shows the percentage space that is used.

Type **df -h .** to show the space sizes in 'human readable' form, e.g. file sizes in megabytes, gigabytes etc. The response will be similar to the following:

```
hristo@hristo-lubuntu18:~$ df . -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        20G  9.4G  9.1G  51% /
```

11.2 The "du" command

The **du** (**d**isk **u**sage) command/utility shows the estimated amount of disk space used by specified files. The current directory is the default directory. Thus the **du** command without options will list the file sizes for files in the current directory. Disk space is normally reported in fixed disk block sizes of 1024 bytes (1k bytes) per block.

Type the **du** command and you will see a display something like the following:

```
hristo@hristo-lubuntu18:~$ du
104     ./labs/LAB1
112     ./labs
248     ./prox
16      ./.AbiSuite
16      ./.eggcups
16      ./.xemacs
8       ./testt
68      ./URLsol
172     ./.mozilla/firefox/4jqxey2v.default/Cache
16      ./.mozilla/firefox/4jqxey2v.default/extensions
```

12. Head and Tail commands

12.1 The "head" command

The **head** command will display a specified number of lines from the beginning of a text file.
For example type:

```
head -2 verse_1
```

This command will display the first two lines of the text file **verse_1**

12.2 The “tail” command

The **tail** command is like the **head** command, but, as you have already guessed, it works from the end of the file, instead of the top of the file.

13. Command History

13.1 The “history” command

The **history** command displays the commands which were previously executed by the shell.

For example, the command **history 3** will display the last three commands.

Also, experiment with the 'up arrow' key (↑) as an easy way to step back to previous commands.

14. The “uniq” command

The **uniq** command is useful to ensure that all lines in a list of lines in a file are unique. The command will, by default, read an input file and then output that file to exclude any adjacent lines that are repeated. There are various options, as described in the **man** pages.

For example:

uniq -u text.txt > textu.txt

This command takes the file **text.txt** as input and outputs all the unique lines to the file **textu.txt**. The **–u** option specifies that only the unique lines are to be outputted.

15. The “cut” command

The **cut** command is useful to cut out columns of data from a file. The command option arguments specify which fields or characters are to be extracted.

For example, at the command line prompt type: **ls –l > logfile**

Assume the content of the **logfile** is now as follows, as might be seen using the command **cat logFile**:

-rw-r--r-- 1 hristo myGroup 198 Apr 3 2007 april
drwxr-xr-x 2 hristo myGroup 4096 Jan 12 11:11 archives
-rwxr-xr-x 1 hristo myGroup 65 Jul 4 2007 busy_wait.txt
drwxr-xr-x 2 hristo myGroup 4096 Jan 12 11:11 limerick
-rwxr-xr-x 1 hristo myGroup 923 Oct 19 2007 load_reduce

Now you want to cut out, or extract, the final column from **logfile**, by counting characters.

Type: **cut -c 51-65 logfile**

The following is now displayed:

april
archives
busy_wait.txt
limerick
load_reduce

Here, the **-c** option defined that the characters in the range between the 51th character and the 65th character were to be extracted.

The **-f** option can be used to extract fields (columns), and the **-d** option can be used to specify the field delimiter. The *tab* is the default field delimiter. However, in the above example the **ls** command output does not use simple delimiters (it uses multiple spaces), so using **awk** (see later) would be more useful to extract columns.

16. The "/proc" virtual filesystem

The Linux kernel also exposes a lot of its internal workings via the **/proc** pseudo-filesystem, which is a very useful way for shell scripts to get access to internal kernel data structures.

The **/proc** directory contains information generated by the Linux kernel about the running processes. Each running process is indicated by its own subdirectory whose name matches the PID for that process. An example of this directory is shown below.

hristo@hristo-lubuntu18:~$ ls /proc

1	11	197	30	41	502	5988	768	899	99	filesystems	mdstat	swaps	
10	112	2	307	42	504	5996	769	9	acpi	fs	meminfo	sys	
100	117	20	31	43	51	6	780	908	asound	interrupts	misc	sysrq-trigger	
1008	12	208	32	44	510	6012	795	912	buddyinfo	iomem	modules	sysvipc	
101	13	21	33	45	512	602	8	913	bus	ioports	mounts	thread-self	
1011	135	212	34	46	53	6022	840	921	cgroups	irq	mtrr	timer_list	
1012	14	22	349	47	532	6024	841	931	cmdline	kallsyms	net	tty	
1013	15	220	3584	476	533	604	850	948	consoles	kcore	pagetypeinfo	uptime	
1028	16	221	36	478	5336	6727	851	951	cpuinfo	keys	partitions	version	
1032	1770	24	37	479	534	6729	855	954	crypto	key-users	sched_debug	version_signature	
1040	1773	25	38	48	5345	7	856	958	devices	kmsg	schedstat	vmallocinfo	
107	18	251	387	481	535	715	862	96	diskstats	kpagecgroup	scsi	vmstat	
1077	19	26	39	482	538	722	864	963	dma	kpagecount	self	zoneinfo	
1081	192	27	3975	484	539	742	878	967	driver	kpageflags	slabinfo		
1085	193	275	4	49	54	744	889	97	execdomains	loadavg	softirqs		
1099	196	28	40	498	598	764	894	98	fb	locks	stat		

The directories with names such as 1, 10, 1040, through 1099 are all of processes. There are other directories containing kernel generation information about the system itself such as acpi, bus, driver, fs, and irq. The remainder of the content of this directory are files about system processes.

The files stored both within **/proc** and in the subdirectories are not true files but instead information about the processes. You will find that all of these have a size of 0.
Some of the files seen in the **/proc** directory also occur within each subdirectory such as cgroups, cmdline, schedstat, and stat. Others are unique for running processes (e.g., environ, maps, mounts, pagemap, and status) and others are unique for the kernel.
Here we examine some of the contents for the processes.

- **cmdline** - the command line instruction that launched this process (if launched from the command line, empty otherwise)

- **environ**—this process's environment variables (if any)
- **fd** - file descriptors (open files)
- **io** - this process' I/O utilization
- **limits** - the limits under which this process was established (e.g., maximum amount of CPU time, maximum file size, and maximum memory utilization)
- **mounts** - mount information
- **cwd** - a link to the current working directory of the process
- **execdomains** - a link to the process' executable file
- **root** - a link to the root directory of the process
-

For the system files, the **cmdline** file stores the kernel's start-up instruction, including all parameters supplied. Other files stored in **/proc** include information on CPU usage, memory usage, average CPU load, file system usage, mounted partitions and their usage, and virtual memory usage.

The information in a **/proc** file is generated on the fly when the file is read. The kernel module that registered a given **/proc** file contains the functions that generate read data and accept write data.
The **/proc** files are a window into the kernel. They provide dynamic information about the state of the system in a way that is easily accessible to user-level tasks and the shell.

Now let's check for particular process of assigned PID, you can get the PID of any running process from **ps command.**

ps aux provides the following output:

```
hristo   1040  0.0  0.1 378812  7532 ?        Ssl  03:47   0:00 /usr/lib/gvfs/gvfs-afc-volume-monitor
hristo   1077  0.0  0.1  59124  5092 ?        S    03:47   0:00 /usr/lib/x86_64-linux-gnu/xfce4/xfconf/xfconfd
hristo   1081  0.0  0.7 468032 28404 ?        Ssl  03:47   0:00 /usr/lib/x86_64-linux-gnu/xfce4/notifyd/xfce4-notifyd
root     1085  0.0  0.2 322288  8492 ?        Ssl  03:47   0:00 /usr/lib/upower/upowerd
hristo   1099  0.0  0.1 368100  7092 ?        Sl   03:47   0:00 /usr/lib/gvfs/gvfsd-trash --spawner :1.4 /org/gtk/gvfs/e
hristo   1770  0.0  0.9 556328 36716 ?        Sl   03:47   0:04 x-terminal-emulator
hristo   1773  0.0  0.1  29808  5172 pts/0    Ss   03:47   0:00 bash
root     3584  0.0  0.0      0     0 ?        I    09:18   0:02 [kworker/0:2]
root     3975  0.0  0.3 373660 13724 ?        Ssl  09:18   0:00 /usr/lib/packagekit/packagekitd
root     5988  0.0  0.0      0     0 ?        I    13:50   0:00 [kworker/u8:0]
root     5996  0.0  0.0      0     0 ?        I    14:06   0:00 [kworker/0:1]
root     6012  0.0  0.1  25660  6016 ?        S    14:06   0:00 /sbin/dhclient -d -q -sf /usr/lib/NetworkManager/nm-dhcp
root     6022  0.0  0.0      0     0 ?        I    14:07   0:00 [kworker/2:1]
root     6732  0.0  0.0      0     0 ?        I    14:46   0:00 [kworker/3:0]
root     6735  0.0  0.0      0     0 ?        R    14:57   0:00 [kworker/u8:3]
root     6737  0.0  0.0      0     0 ?        I    15:01   0:00 [kworker/1:0]
root     6812  0.0  0.0      0     0 ?        I    15:07   0:00 [kworker/2:0]
```

```
root      6813  0.0  0.0      0     0 ?        I    15:07   0:00 [kworker/u8:1]
root      6817  0.0  0.0      0     0 ?        I    15:07   0:00 [kworker/3:2]
hristo    7302  0.0  0.0  28572  3864 pts/0    S+   15:07   0:00 man proc
hristo    7311  0.0  0.0  28308   940 pts/0    S+   15:07   0:00 man proc
hristo    7312  0.0  0.0  16956  1016 pts/0    S+   15:07   0:00 pager
hristo    7320  0.0  0.1  29644  5084 pts/1    Ss   15:10   0:00 bash
hristo    7331 72.6  7.7 2235728 312640 ?      Sl   15:11   0:09 /usr/lib/firefox/firefox
hristo    7538 10.2  3.4 1543840 140592 ?      Sl   15:11   0:01 /usr/lib/firefox/firefox -contentproc -childID 1 -isForB
hristo    7624  3.4  2.5 1503792 104344 ?      Sl   15:11   0:00 /usr/lib/firefox/firefox -contentproc -childID 3 -isForB
hristo    7696 45.7  9.5 1937736 386828 ?      Sl   15:11   0:04 /usr/lib/firefox/firefox -contentproc -childID 4 -isForB
hristo    7764  1.6  1.9 1471240 77056 ?       Sl   15:11   0:00 /usr/lib/firefox/firefox -contentproc -childID 5 -isForB
hristo    7813  0.0  0.0  46772  3764 pts/1    R+   15:11   0:00 ps –aux
```

Now check the highlighted process with PID=7331, you can check that there is entry for this process in **/proc** file system.

hristo@hristo-lubuntu18:~$ ls -l /proc/7331/

```
total 0
dr-xr-xr-x  2 hristo hristo 0 Feb  5 15:15 attr
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 autogroup
-r--------  1 hristo hristo 0 Feb  5 15:15 auxv
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 cgroup
--w-------  1 hristo hristo 0 Feb  5 15:15 clear_refs
-r--r--r--  1 hristo hristo 0 Feb  5 15:11 cmdline
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 comm
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 coredump_filter
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 cpuset
lrwxrwxrwx  1 hristo hristo 0 Feb  5 15:15 cwd -> /home/hristo
-r--------  1 hristo hristo 0 Feb  5 15:15 environ
lrwxrwxrwx  1 hristo hristo 0 Feb  5 15:11 exe -> /usr/lib/firefox/firefox
dr-x------  2 hristo hristo 0 Feb  5 15:11 fd
dr-x------  2 hristo hristo 0 Feb  5 15:15 fdinfo
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 gid_map
-r--------  1 hristo hristo 0 Feb  5 15:15 io
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 limits
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 loginuid
dr-x------  2 hristo hristo 0 Feb  5 15:15 map_files
-r--r--r--  1 hristo hristo 0 Feb  5 15:11 maps
-rw-------  1 hristo hristo 0 Feb  5 15:15 mem
-r--r--r--  1 hristo hristo 0 Feb  5 15:11 mountinfo
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 mounts
-r--------  1 hristo hristo 0 Feb  5 15:15 mountstats
dr-xr-xr-x  5 hristo hristo 0 Feb  5 15:11 net
dr-x--x--x  2 hristo hristo 0 Feb  5 15:11 ns
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 numa_maps
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 oom_adj
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 oom_score
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 oom_score_adj
-r--------  1 hristo hristo 0 Feb  5 15:15 pagemap
-r--------  1 hristo hristo 0 Feb  5 15:15 patch_state
-r--------  1 hristo hristo 0 Feb  5 15:15 personality
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 projid_map
lrwxrwxrwx  1 hristo hristo 0 Feb  5 15:15 root -> /
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 sched
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 schedstat
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 sessionid
-rw-r--r--  1 hristo hristo 0 Feb  5 15:15 setgroups
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 smaps
-r--r--r--  1 hristo hristo 0 Feb  5 15:15 smaps_rollup
-r--------  1 hristo hristo 0 Feb  5 15:15 stack
-r--r--r--  1 hristo hristo 0 Feb  5 15:11 stat
```

-r--r--r-- 1 hristo hristo 0 Feb 5 15:15 statm
-r--r--r-- 1 hristo hristo 0 Feb 5 15:11 status
-r-------- 1 hristo hristo 0 Feb 5 15:15 syscall
dr-xr-xr-x 62 hristo hristo 0 Feb 5 15:11 task
-r--r--r-- 1 hristo hristo 0 Feb 5 15:15 timers
-rw-rw-rw- 1 hristo hristo 0 Feb 5 15:15 timerslack_ns
-rw-r--r-- 1 hristo hristo 0 Feb 5 15:15 uid_map
-r--r--r-- 1 hristo hristo 0 Feb 5 15:15 wchan

Checking the status of the firefox process:

hristo@hristo-lubuntu18:~$ cat /proc/7331/status

Name: firefox
Umask: 0002
State: S (sleeping)
Tgid: 7331
Ngid: 0
Pid: 7331
PPid: 908
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 512
Groups: 4 24 27 30 46 114 116 122 999 1000
NStgid: 7331
NSpid: 7331
NSpgid: 780
NSsid: 780
VmPeak: 2348516 kB
VmSize: 2312840 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 396960 kB
VmRSS: 275916 kB
RssAnon: 143360 kB
RssFile: 123060 kB
RssShmem: 9496 kB
VmData: 395044 kB
VmStk: 132 kB
VmExe: 200 kB
VmLib: 210620 kB
VmPTE: 1772 kB
VmSwap: 0 kB
HugetlbPages: 0 kB
CoreDumping: 0
Threads: 65
SigQ: 0/15541
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000001000
SigCgt: 0000000f800044ff
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
NoNewPrivs: 0
Seccomp: 0
Speculation_Store_Bypass: vulnerable

Cpus_allowed: f
Cpus_allowed_list: 0-3
Mems_allowed:
00000000,00000001
Mems_allowed_list: 0
voluntary_ctxt_switches: 24679
nonvoluntary_ctxt_switches: 36991

END END END END END