UNIX Programming (18CS56)

Module 1

(I) Introduction

A. Operating System (OS):

- *Definition* – An operating system (OS) is a software that manages the computer's hardware and provides a convenient and safe environment for running programs and performing various tasks.
- It is basically an interface between user-written programs and hardware resources.
- It is the first program to be loaded onto the memory when the computer is switched on and it remains active as long as the machine is running.
- There are several OS in the market, as of today. Some examples are – Microsoft Windows, MS-DOS, UNIX, Linux, macOS.

B. UNIX Operating System:

- One of the oldest operating systems, it was built earlier than MS-DOS and Microsoft Windows.
- The UNIX OS was developed in the 1970's at AT&T Bell Labs by Ken Thompson, Dennis Ritchie and other fellow researchers.
- A user can interact with the UNIX OS via a command interpreter known as the "*shell*."
- The shell considers any word/character input as a command, and gives the respective output.
  If the command is valid, then the output will be with respect to the input provided.
  If the command is invalid, the shell may or may not throw an error.
- The main power/advantage of UNIX lies in the ability to combine multiple commands to perform various functions.

(II) UNIX Architecture

- As seen in Figure 1, the UNIX architecture follows a layered structure, like that of an onion.
- The innermost layer consists of the computer hardware which can only be accessed by the kernel.
- The kernel is the next layer which acts as a bridge between the shell and the hardware.
- The shell is the outermost layer and the user interacts with this layer, and inputs commands into it.

- The shell is accompanied by several application programs, compilers, text processors, databases and other software.

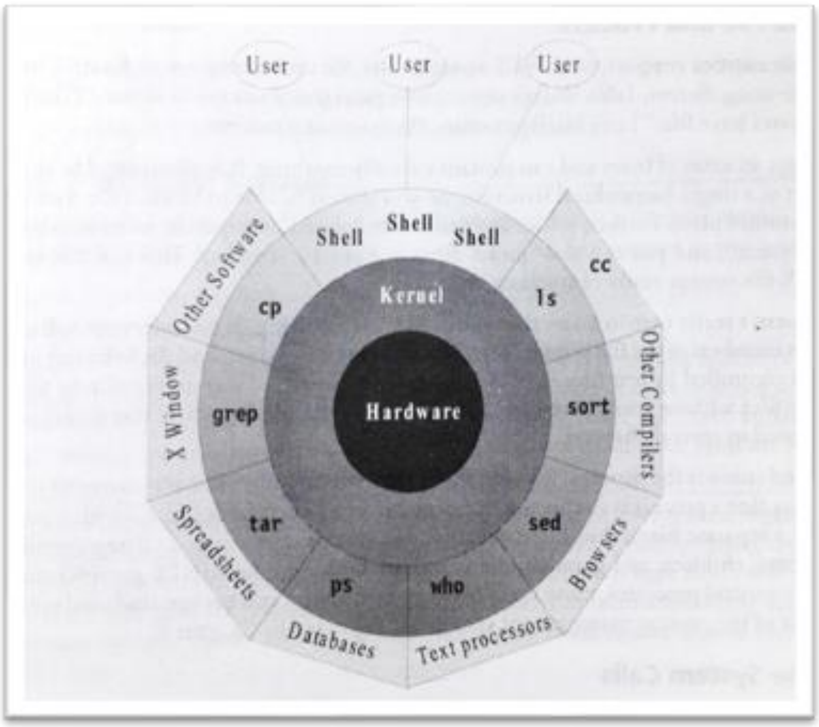
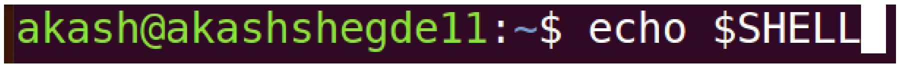
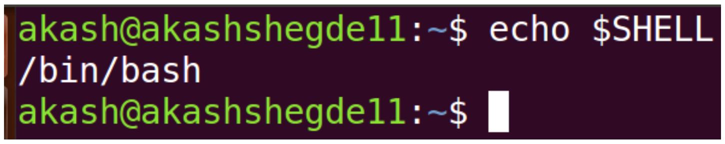


Figure 1. Architecture of UNIX: The Kernel-Shell Relationship

- *Kernel*:
  - The kernel is the core of the UNIX operating system.
  - It mainly consists of a collection of routines written in the C language.
  - It is loaded onto the main memory as soon as the system is booted up.
  - The kernel communicates directly with the hardware.
  - If a user wishes to access hardware, the kernel performs the job on behalf of the user through execution of "*system calls.*"
  - The kernel does all sorts of housekeeping work even when there are no active requests – it performs memory management, process scheduling, assigns the priorities of processes, amongst other activities.
- *Shell*:
  - The shell is the outer part of the UNIX operating system.
  - The main function of the shell lies in translating user input commands into action – it is the *command interpreter* of the UNIX operating system.
  - It is responsible for being the interface between the external user and the kernel.

- One UNIX kernel can have multiple shells (Bourne shell, C shell, Korn shell, Bash shell, Z shell), and the user can switch between these shells as and when he/she wishes to do so.
- The shell also contains multiple *prompts* for the user – **$**, **%**, **#**
  $ - main prompt for any user to input commands, restricted by the system administrator.
  % - similar to **$**, usually present in older operating systems.
  # - root access prompt, able to perform administrative functions.
- The shell which the terminal is using can be checked using the ***echo*** command with the ***$SHELL*** environment variable.
  The usage is given as follows:
  

  ```
  akash@akashshegde11:~$ echo $SHELL
  ```

  The output for the above ***echo*** command is as follows, which indicates the Bash shell being used:
  

  ```
  akash@akashshegde11:~$ echo $SHELL
  /bin/bash
  akash@akashshegde11:~$
  ```

- *File*:
  - A file in UNIX is defined as an array of bytes that can contain anything.
  - A file is related to another file by being a part of single hierarchical structure.
  - Files are contained in directories.
  - However, UNIX does not care about the type of file – all directories and connected devices are considered to be a part of the file system.
  - The dominant file type in UNIX systems is the *text* file.
- *Process*:
  - A process in UNIX is basically the name given to a file when it is executed as a program.
  - Similar to the behavior of files, processes also belong to a hierarchical tree structure.
  - Processes exhibit similar characteristics as that of living organisms – they contain *parent* processes, *child* processes; they are *born* and they *die*; they can also become *orphan* processes and *zombie* processes.
- *System Calls*:
  - The UNIX OS consists of more than a thousand commands, and all of these commands use *system calls* to communicate with the kernel.
  - A system call is a request sent to the OS to do something on behalf of the user’s program.
  - The major advantage of using UNIX-based systems is that all of them use the same system calls.

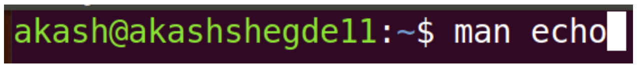
- These system calls are built into the kernel of the OS itself, thereby making it extremely easy to port them from UNIX-based machine to another.
- C programmers on a Windows machine use the standard library functions in order to make lengthy requests to the OS, however the same programmers use simple system calls that directly communicate with the kernel in UNIX. Example: *write* (system call) in UNIX vs. fprintf (standard library function) in Windows.

(III) Features of UNIX

- UNIX operating systems has several features and supports a wide variety of them.
- Some of the main features of UNIX are as follows:
  - Multiuser System
  - Multitasking System
  - Building-Block Approach
  - UNIX Toolkit
  - Pattern Matching
  - Programming Facility
  - Documentation
- *Multiuser System:*
  - The fundamental view of the UNIX OS is that it allows for *multiprogramming*.
  - This means that multiple programs can run simultaneously and compete for the available resources.
  - This can occur in two situations – one where there are *multiple users* running separate single jobs, and another where a *single user* is running multiple jobs.
  - Unlike Windows, system resources are shared between all users in UNIX. This allows a lot of savings with respect to hardware and licensed software.
  - The existence of a multiuser system is actually a very clever illusion done by the UNIX OS.
  - It breaks up a unit of time (in the order of nanoseconds) into multiple segments and allots these segments to all the users of the system.
- *Multitasking System*:
  - UNIX allows for a single user to run multiple tasks concurrently.
  - The kernel is designed such that it can handle a user's multiple needs at the same time – it can edit a file, print a file, send an email and browse the Internet all together.
  - This is done by running one job in the *foreground*, whereas the remaining jobs are running in the *background*.

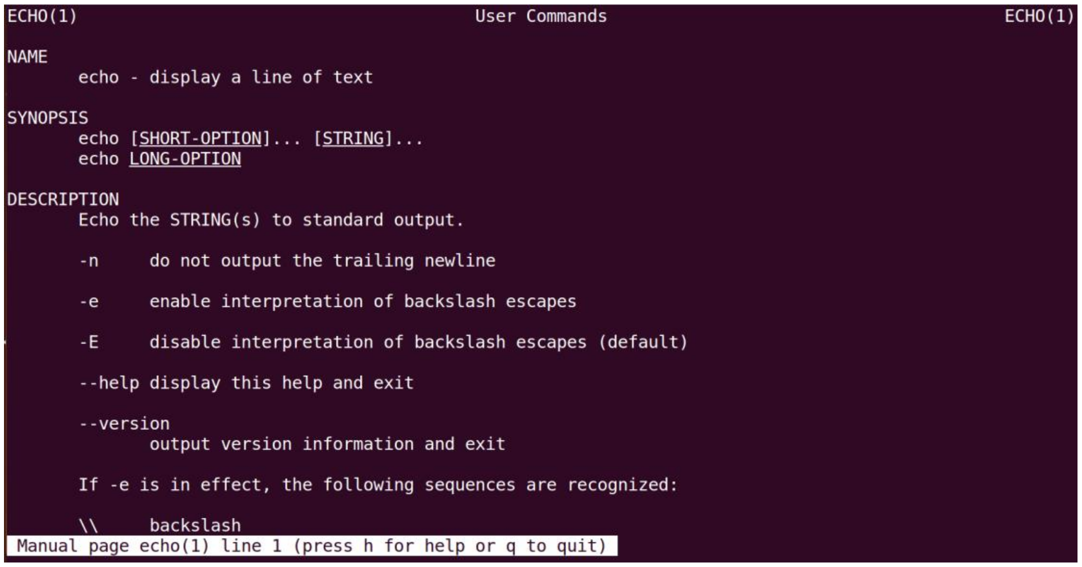
  - The user can switch between multiple tasks effortlessly, as the kernel switches between foreground and background jobs almost instantly.
- *Building-Block Approach*:
  - UNIX follows a simplistic approach to performing tasks/jobs.
  - It consists of a few hundred commands which perform simple jobs, rather than packing too many features into a few tools and invoking complex commands.
  - UNIX allows and encourages combining simple commands to perform powerful functions and tasks – these are achieved through the use of *pipes* and *filters*.
  - It believes in building commands that can handle specialized functions than try to solve multiple problems at once.
  - Another main feature of the building-block approach of UNIX is that it allows the output of one tool to act as input to another tool.
- *UNIX Toolkit*:
  - UNIX OS provides a huge and diverse range of tools, which can be general purpose tools, text manipulation utilities, compilers and interpreters, networked applications and system administration tools.
  - The toolkit also allows for switching between shells, if and when required.
  - New tools are added to the UNIX toolkit with each UNIX release, whereas old tools are modified or removed altogether.
- *Pattern Matching*:
  - One of the main features of UNIX is that it allows for sophisticated pattern matching.
  - There are several special characters in UNIX that are used for pattern matching.
  - These special characters are called *metacharacters*, and are given as follows: ***? [ ] ' " \ $; & ( ) | ^ < > newline space tab***
  - UNIX also allows for *regular expressions* (RE) which are special expressions formed from the metacharacter set given above.
- *Programming Facility*:
  - UNIX is designed to support all types of programming constructs, such as control structures, loops and variables.
  - Shell scripts can be written in UNIX, which are programs that invoke UNIX commands.
  - Programs can be written to control and automate certain system functions, as well as customize them to the user’s liking.
  - This also provides a useful career opportunity as a *system administrator*.
- *Documentation*:
  - Extensive documentation regarding all the commands is given by an in-built command – ***man*** command.

- It is a detailed manual which acts as a reference for commands and their respective configurations.
- Example usage of ***man*** command:



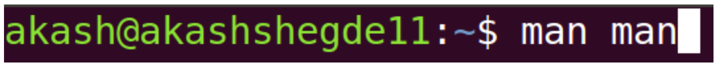
```
akash@akashshegde11:~$ man echo
```

Output of the above ***man*** command:



```
ECHO(1)                          User Commands                         ECHO(1)

NAME
       echo - display a line of text

SYNOPSIS
       echo [SHORT-OPTION]... [STRING]...
       echo LONG-OPTION

DESCRIPTION
       Echo the STRING(s) to standard output.

       -n     do not output the trailing newline

       -e     enable interpretation of backslash escapes

       -E     disable interpretation of backslash escapes (default)

       --help display this help and exit

       --version
              output version information and exit

       If -e is in effect, the following sequences are recognized:

       \\     backslash
 Manual page echo(1) line 1 (press h for help or q to quit)
```

- The ***man*** command can also be used with itself, as it is inherently a command too. Usage is as follows:



```
akash@akashshegde11:~$ man man
```

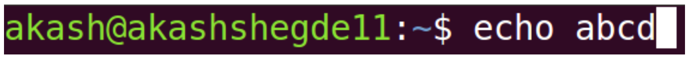
- Several websites, articles and videos are hosted online with respect to the documentation of UNIX commands and features.
- Some websites even host online terminals for practicing UNIX commands.

(IV) POSIX and the Single UNIX Specification

- POSIX stands for Portable Operating System Interface for Computer Environments.
- It is a group of standards that are specified by the IEEE Computer Society for maintaining compatibility between UNIX-based operating systems.
- There are mainly two specifications that are historically important –
  i. POSIX.1 deals with the UNIX system calls.
  ii. POSIX.2 deals with the shell and other UNIX utilities.
- The Single UNIX Specification (SUS) has been standardized to provide easy portability between POSIX-compliant systems.
- The approach of "write once, adopt anywhere" has been made when compiling the SUS.

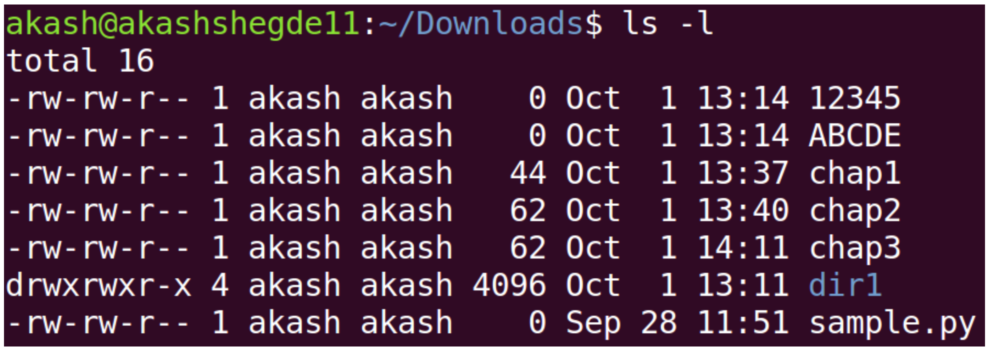
(V) Command Structure

- Commands in UNIX follow a simple syntax, where a *command* is written first and is immediately followed by an *argument* to it.
- An example ***echo*** command with an argument is shown below:



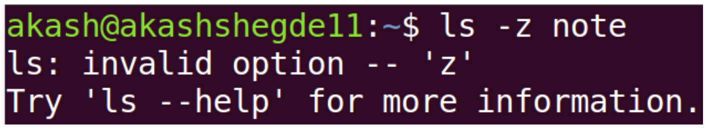
```
akash@akashshegde11:~$ echo abcd
```

- Here, the command and the argument are separated by a *whitespace*.
- A whitespace is generally used as a *delimiter* to separate the command and the argument in UNIX.
- The shell provides permission to the user to use multiple whitespaces between the command and the argument, in order to clearly distinguish words.
- There can be a wide range of arguments in UNIX – options, expressions, instructions, filenames can all be used as arguments to the commands.
- An *option* is a special type of argument.
- It is represented by the **-** (minus) sign and is frequently used with commands to display some special outputs.
- For example, the command to list the files in a directory is called the ***ls*** command. This command utilizes an option ***-l*** to list the files and their attributes in a detailed manner. The command, its option and the resulting output are shown below:



```
akash@akashshegde11:~/Downloads$ ls -l
total 16
-rw-rw-r-- 1 akash akash    0 Oct  1 13:14 12345
-rw-rw-r-- 1 akash akash    0 Oct  1 13:14 ABCDE
-rw-rw-r-- 1 akash akash   44 Oct  1 13:37 chap1
-rw-rw-r-- 1 akash akash   62 Oct  1 13:40 chap2
-rw-rw-r-- 1 akash akash   62 Oct  1 14:11 chap3
drwxrwxr-x 4 akash akash 4096 Oct  1 13:11 dir1
-rw-rw-r-- 1 akash akash    0 Sep 28 11:51 sample.py
```

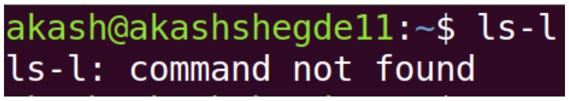
- Options are prefixed by the **-** (minus) sign in order to distinguish the special argument from the filenames.
- The list of options is pre-determined by the shell and built in to it, whereas normal arguments do not have any pre-determined existence or significance in the shell.
- When a command is used with the wrong/undefined option, it yields an error. An example with the ***ls*** command is shown below:



```
akash@akashshegde11:~$ ls -z note
ls: invalid option -- 'z'
Try 'ls --help' for more information.
```

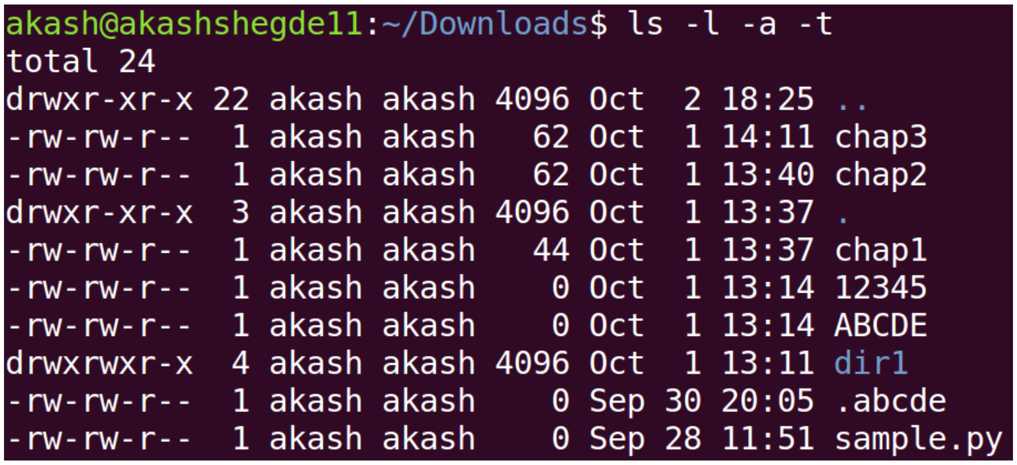
- When an option is used without providing whitespace between command and argument, it yields an error.

An example with the ***ls*** command is shown below:



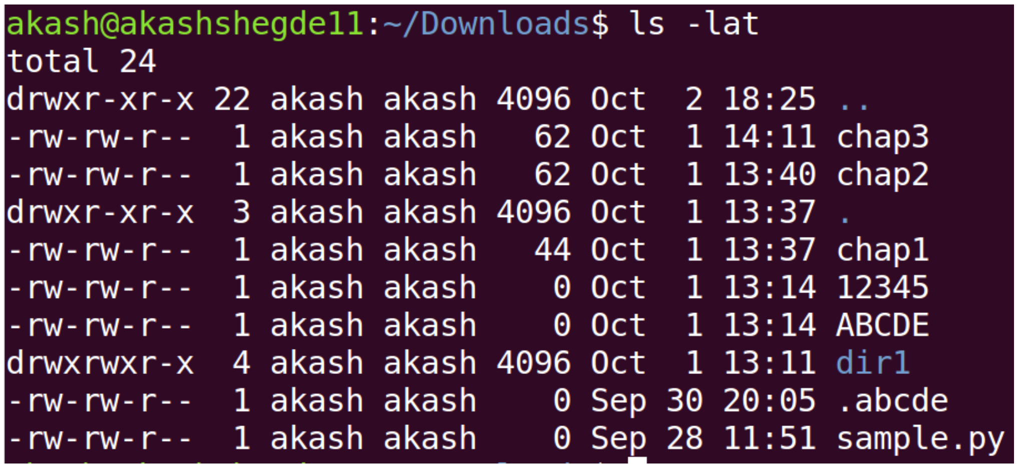
```
akash@akashshegde11:~$ ls-l
ls-l: command not found
```

- Multiple options can be provided to the command on the same line.
  An example with the ***ls*** command is shown below:



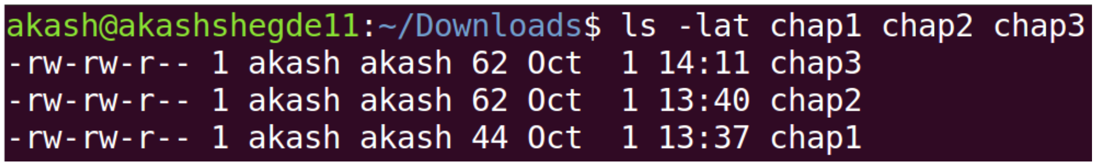
```
akash@akashshegde11:~/Downloads$ ls -l -a -t
total 24
drwxr-xr-x 22 akash akash 4096 Oct  2 18:25 ..
-rw-rw-r--  1 akash akash   62 Oct  1 14:11 chap3
-rw-rw-r--  1 akash akash   62 Oct  1 13:40 chap2
drwxr-xr-x  3 akash akash 4096 Oct  1 13:37 .
-rw-rw-r--  1 akash akash   44 Oct  1 13:37 chap1
-rw-rw-r--  1 akash akash    0 Oct  1 13:14 12345
-rw-rw-r--  1 akash akash    0 Oct  1 13:14 ABCDE
drwxrwxr-x  4 akash akash 4096 Oct  1 13:11 dir1
-rw-rw-r--  1 akash akash    0 Sep 30 20:05 .abcde
-rw-rw-r--  1 akash akash    0 Sep 28 11:51 sample.py
```

- These options can be combined together under a single - (minus) sign to produce the same output as above.
  An example with the ***ls*** command is shown below:



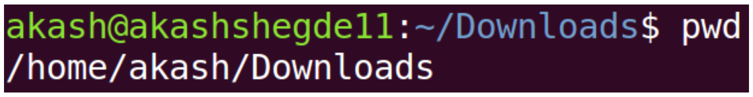
```
akash@akashshegde11:~/Downloads$ ls -lat
total 24
drwxr-xr-x 22 akash akash 4096 Oct  2 18:25 ..
-rw-rw-r--  1 akash akash   62 Oct  1 14:11 chap3
-rw-rw-r--  1 akash akash   62 Oct  1 13:40 chap2
drwxr-xr-x  3 akash akash 4096 Oct  1 13:37 .
-rw-rw-r--  1 akash akash   44 Oct  1 13:37 chap1
-rw-rw-r--  1 akash akash    0 Oct  1 13:14 12345
-rw-rw-r--  1 akash akash    0 Oct  1 13:14 ABCDE
drwxrwxr-x  4 akash akash 4096 Oct  1 13:11 dir1
-rw-rw-r--  1 akash akash    0 Sep 30 20:05 .abcde
-rw-rw-r--  1 akash akash    0 Sep 28 11:51 sample.py
```

- Arguments can also be filenames.
  When this occurs, the command takes input from the specified file.
- The filename is usually kept as the last argument in the command line.
- Even multiple filenames can be passed as arguments to the command.
  An example with the ***ls*** command is shown below:



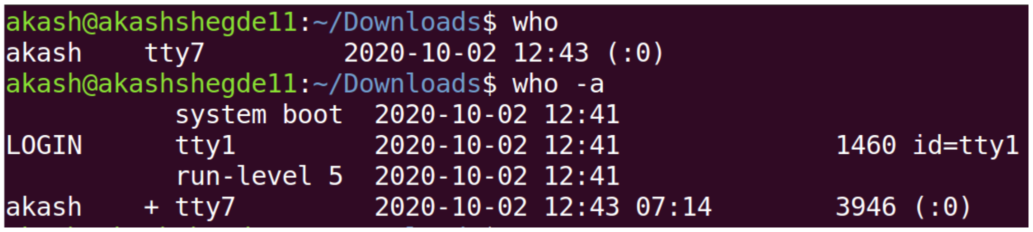
```
akash@akashshegde11:~/Downloads$ ls -lat chap1 chap2 chap3
-rw-rw-r-- 1 akash akash 62 Oct  1 14:11 chap3
-rw-rw-r-- 1 akash akash 62 Oct  1 13:40 chap2
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

- Overall, the *command line* consists of the *command* and its associated *arguments/options*.
- There are a few *exceptions* to the above stated command structure:
  - There exist commands which have no arguments.
    The ***pwd*** command to print the working directory is one such command.



```
akash@akashshegde11:~/Downloads$ pwd
/home/akash/Downloads
```

  - Some commands may or may not have arguments.
    The ***who*** command to display the users on the system is one such command.

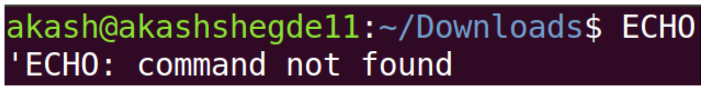


```
akash@akashshegde11:~/Downloads$ who
akash    tty7         2020-10-02 12:43 (:0)
akash@akashshegde11:~/Downloads$ who -a
           system boot  2020-10-02 12:41
LOGIN      tty1         2020-10-02 12:41             1460 id=tty1
           run-level 5  2020-10-02 12:41
akash    + tty7         2020-10-02 12:43 07:14       3946 (:0)
```

  - Some commands can run without arguments, with only their options, with only filename arguments, or using a combination of both.
    The ***ls*** command is the best example for it and a few outputs have already been displayed for this command.
  - There are a few commands which compulsorily need to have an option specified.
    The ***cut*** command to cut sections of outputs is one such command.
  - There are a few commands which have expressions as arguments.
    The ***grep*** command used for pattern matching is one such command.
  - There also exist commands which have a set of instructions as arguments.
    The ***sed*** command to edit and filter text is one such command.
  - Finally, there are also some commands which have an entire program as arguments.
    The ***awk*** and ***perl*** scripting commands are good examples.

(VI) Locating Commands

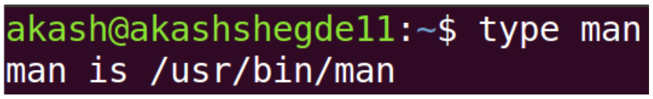
- UNIX commands are all case-sensitive and are all in lowercase.
- Violation of this rule will produce an error in the output.



```
akash@akashshegde11:~/Downloads$ ECHO
'ECHO: command not found
```

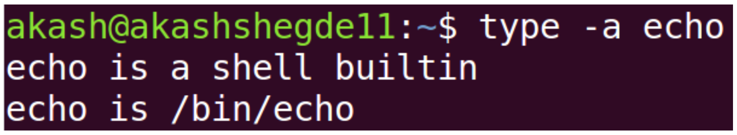
- These commands are simply UNIX files that contain programs written in the C programming language.
- These commands are stored in *directories* in the UNIX file system.

- It is important to know the location of these commands.
  This can be done using the ***type*** command in UNIX, as shown below:



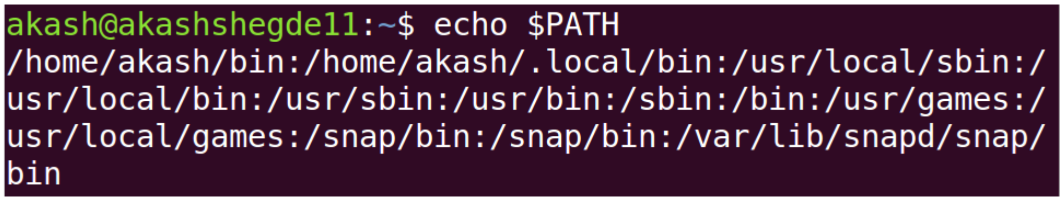
```
akash@akashshegde11:~$ type man
man is /usr/bin/man
```

  The output gives the location of the command's program in the file system.
- The ***type*** command only looks in directories specified in the **PATH** environment variable.
- The ***type*** command comes with the ***–a*** option to check for aliases of commands, which may exist in multiple locations.
  The ***echo*** command is both an in-built command as well as a separate command with its own executable program.



```
akash@akashshegde11:~$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

- The **PATH** environment variable specifies a set of directories where programs are stored.
  The output of the echo command for the **PATH** variable is as follows:



```
akash@akashshegde11:~$ echo $PATH
/home/akash/bin:/home/akash/.local/bin:/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
usr/local/games:/snap/bin:/snap/bin:/var/lib/snapd/snap/
bin
```

- The executables and programs are delimited by a **:** (colon) in the **PATH** variable.
- The / (forward slash) stands for the root directory.
- The delimiter used in UNIX for **PATH** variable is **:** (colon), whereas the same used in Windows is **;** (semi-colon).
- All the essential UNIX commands are stored in the directories ***/bin*** and ***/usr/bin***

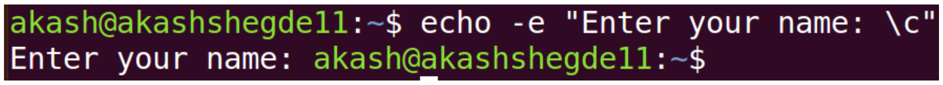
(VII) Internal and External Commands:

- An *external* command is a program/file that has an independent existence in the ***/bin*** (or ***/usr/bin***) directory.
  The ***man*** command is a prime example for an external command, as we have seen its output in the previous section.
- An *internal* command is a built-in command that is not stored as a separate file in the UNIX file system.
  The ***echo*** command is a good example for an internal command.
  It used to be an external command in earlier versions of UNIX, but was then integrated into the shell as an internal command.

- The *shell* is a special type of external command, as it contains its own set of internal commands.
- If a command exists as both internal command of shell and external to shell (in ***/bin*** or ***/usr/bin***), the shell will give priority to its own internal command and execute it first.
  This has been the case for the ***echo*** command in modern UNIX systems.

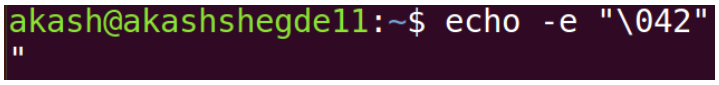
(VIII) Basic UNIX Commands:

- There are several UNIX commands that are used for basic utilities and functions.
- ***echo*** command –
  echo command is used to display a line of text/string that is passed as an argument to the command.
- It is often used in shell scripts to display diagnostic messages on the terminal.
- It is also used to issue prompts to take user input.
- As mentioned in the previous section, ***echo*** was an external command in earlier versions of UNIX, but has now been integrated into the shell.
- An *escape sequence* is generally a two-character string beginning with a \ (backslash).
- It is usually placed at the end of a string to act as an input prompt.
- In Bash shell, ***-e*** option needs to be provided in order to use the escape sequence.
  An example with the ***echo*** command is given below:



```
akash@akashshegde11:~$ echo -e "Enter your name: \c"
Enter your name: akash@akashshegde11:~$
```

  The **\c** escape sequence prints the cursor on the same line, as can be seen above.
- Octal values of ASCII characters can also be used as escape sequences.
- The octal value should be preceded by **\0**, and as such,***-e*** option has to be used with the ***echo*** command to display these characters.
  An example to print " (double quotes) has been shown below:



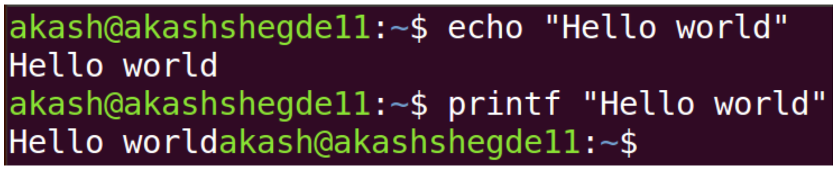
```
akash@akashshegde11:~$ echo -e "\042"
"
```

- Some of the major escape sequences are given in the table below:

Escape Sequence	**Significance**
\a	Bell
\b	Backspace
\c	No newline (cursor in same line)
\f	Form feed
\n	Newline
\r	Carriage return

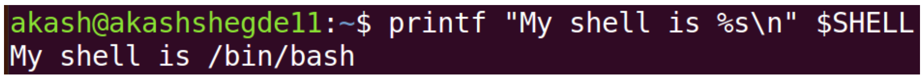
\t	Tab
\v	Vertical tab
\\	Backslash
\0*n*	ASCII character represented by octal value *n*, where $n \leq 377$

- ***printf*** command –
  printf is an alternative to ***echo*** command, and is used to display line of text / strings.
- Most shells use ***printf*** as an external command, only Bash shell has it built-in.
- One major difference between ***printf*** and ***echo*** is that ***printf*** does not automatically insert new line unless **\n** escape sequence is specified, unlike ***echo***.



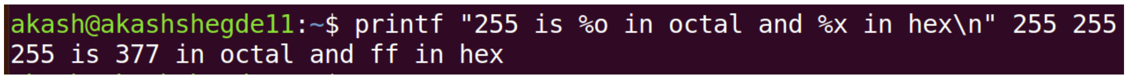
```
akash@akashshegde11:~$ echo "Hello world"
Hello world
akash@akashshegde11:~$ printf "Hello world"
Hello worldakash@akashshegde11:~$
```

- ***printf*** uses formatted strings in UNIX just like in C language.
  An example with the ***%s*** format specifier is given below:



```
akash@akashshegde11:~$ printf "My shell is %s\n" $SHELL
My shell is /bin/bash
```

- Multiple format specifiers can be used in a single ***printf*** command.
  Care should be taken to specify as many arguments as there are format specifiers in the ***printf*** command, and these should be in the right order.

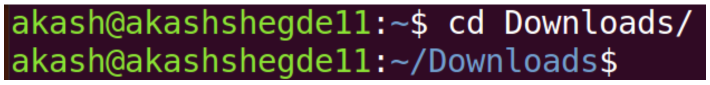


```
akash@akashshegde11:~$ printf "255 is %o in octal and %x in hex\n" 255 255
255 is 377 in octal and ff in hex
```

- Some of the major format specifiers are given in the table below:

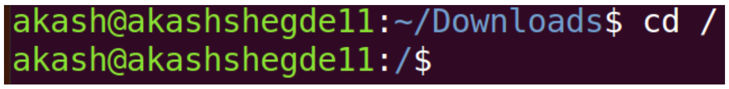
Format Specifier	**Significance**
%s	String
%30s	String, but printed in a space 30 characters wide
%d	Decimal integer
%6d	Decimal integer, but printed in a space 6 characters wide
%o	Octal integer
%x	Hexadecimal integer
%f	Floating point number

- ***cd*** command –
  cd command is used to change the current working directory.



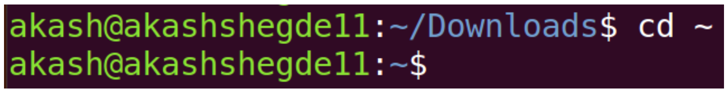
```
akash@akashshegde11:~$ cd Downloads/
akash@akashshegde11:~/Downloads$
```

- Changing to *root* directory can be done with ***cd /***



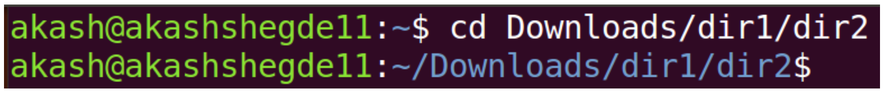
```
akash@akashshegde11:~/Downloads$ cd /
akash@akashshegde11:/$
```

- Changing to *home* directory can be done either with ***cd ~*** or just ***cd***



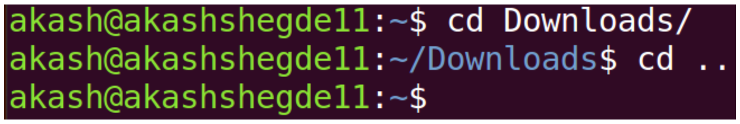
```
akash@akashshegde11:~/Downloads$ cd ~
akash@akashshegde11:~$
```

- Changing to some directory can be done with ***cd dir1/dir2/dir3***



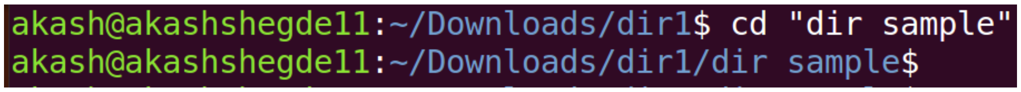
```
akash@akashshegde11:~$ cd Downloads/dir1/dir2
akash@akashshegde11:~/Downloads/dir1/dir2$
```

- Changing to the parent directory of the current directory can be done with ***cd ..***



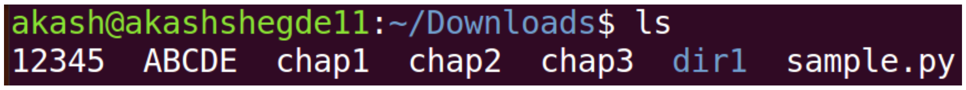
```
akash@akashshegde11:~$ cd Downloads/
akash@akashshegde11:~/Downloads$ cd ..
akash@akashshegde11:~$
```

- Changing to a directory which has spaces in its name can be done either with ***cd "dir name"*** or ***cd dir\ name***



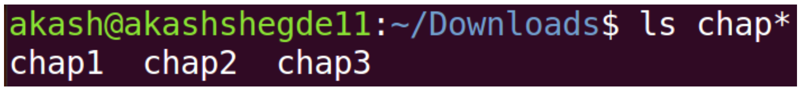
```
akash@akashshegde11:~/Downloads/dir1$ cd "dir sample"
akash@akashshegde11:~/Downloads/dir1/dir sample$
```

- ***ls*** command –
  ls command is used to list files in a directory.
- The default listing of files follows the arrangement as follows – files are arranged alphabetically with uppercase filenames having precedence over lowercase filenames. This type of arrangement is known as the *ASCII collating sequence*.
- Sample output of ***ls*** command is shown below:



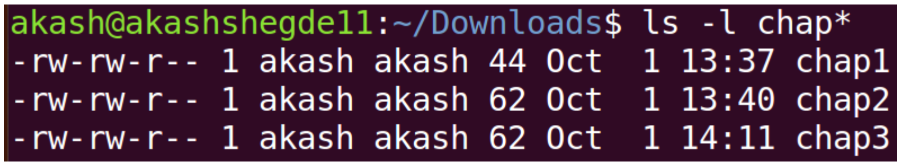
```
akash@akashshegde11:~/Downloads$ ls
12345  ABCDE  chap1  chap2  chap3  dir1  sample.py
```

- Listing files with similar filenames uses ***** for matching and is shown below:



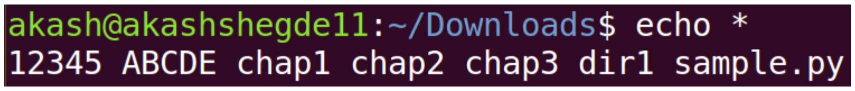
```
akash@akashshegde11:~/Downloads$ ls chap*
chap1  chap2  chap3
```

- Listing files with their detailed descriptions uses ***–l*** option and is shown below:



```
akash@akashshegde11:~/Downloads$ ls -l chap*
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
-rw-rw-r-- 1 akash akash 62 Oct  1 13:40 chap2
-rw-rw-r-- 1 akash akash 62 Oct  1 14:11 chap3
```

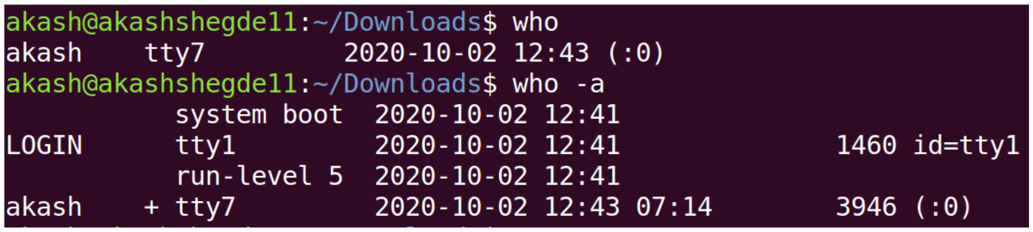
- ***echo*** command can also be used to list files in a directory. This is shown below:



```
akash@akashshegde11:~/Downloads$ echo *
12345 ABCDE chap1 chap2 chap3 dir1 sample.py
```

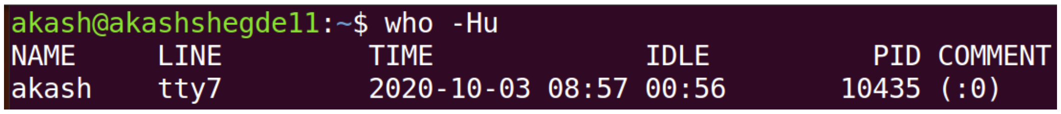
- ***who*** command –

  who command provides an account of all the users who are logged on to the system.



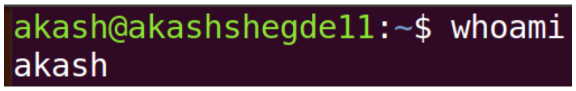
```
akash@akashshegde11:~/Downloads$ who
akash    tty7         2020-10-02 12:43 (:0)
akash@akashshegde11:~/Downloads$ who -a
           system boot  2020-10-02 12:41
LOGIN      tty1         2020-10-02 12:41             1460 id=tty1
           run-level 5  2020-10-02 12:41
akash    + tty7         2020-10-02 12:43 07:14       3946 (:0)
```

- Detailed descriptions with all the headers can be obtained by using ***–Hu*** option with ***who*** command.



```
akash@akashshegde11:~$ who -Hu
NAME     LINE         TIME             IDLE          PID COMMENT
akash    tty7         2020-10-03 08:57 00:56       10435 (:0)
```

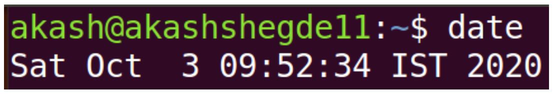
- The user who is currently on the active account can be found out by using ***whoami***



```
akash@akashshegde11:~$ whoami
akash
```

- ***date*** command –

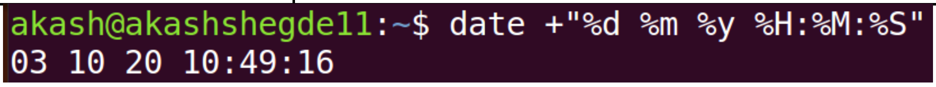
  date command is used to display the system date.



```
akash@akashshegde11:~$ date
Sat Oct  3 09:52:34 IST 2020
```

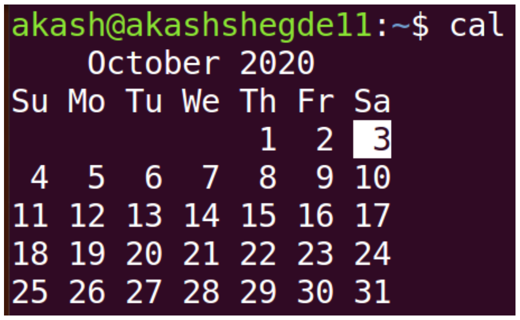
- UNIX systems have an internal clock running since 01 January 1970. This is known as *the Epoch*.
- A 32-bit counter stores these seconds.
- Some of the major format specifiers for ***date*** command are given in the table below:

Format Specifier	**Significance**
+%d	Date
+%m	Month number
+%h	Month name
+%y	Year
+%H	Hour
+%M	Minute
+%S	Seconds
+%D	Date in the format *dd/mm/yyyy*
+%T	Time in the format *hh:mm:ss*
+"%d %m %y %H:%M:%S"	Multiple format specifiers together (enclose within double quotes, use single + symbol before the quotes)



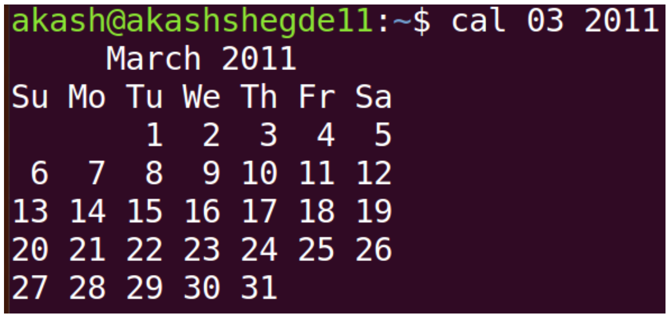
```
akash@akashshegde11:~$ date +"%d %m %y %H:%M:%S"
03 10 20 10:49:16
```

- ***passwd*** command –
  passwd command is used to change the user password.
- The command expects 3 responses from the user – old password, new password and re-entering the new password.
- The password entered by the user will be encrypted by the system.
- The encrypted password will be stored in the file ***/etc/shadow***
- Some general password framing rules are given as follows:
  - Don't choose a password similar to the old one.
  - Don't use commonly used names like names of friends, relatives, pets and so on.
  - Use a mix of alphabetic and numeric characters.
  - Don't write down the password in an easily accessible document.
  - Change the password regularly.
- ***cal*** command –
  cal command is used to see the calendar of any specific month or a complete year.
- The default output of the ***cal*** command will be the calendar of the current month, as shown below.



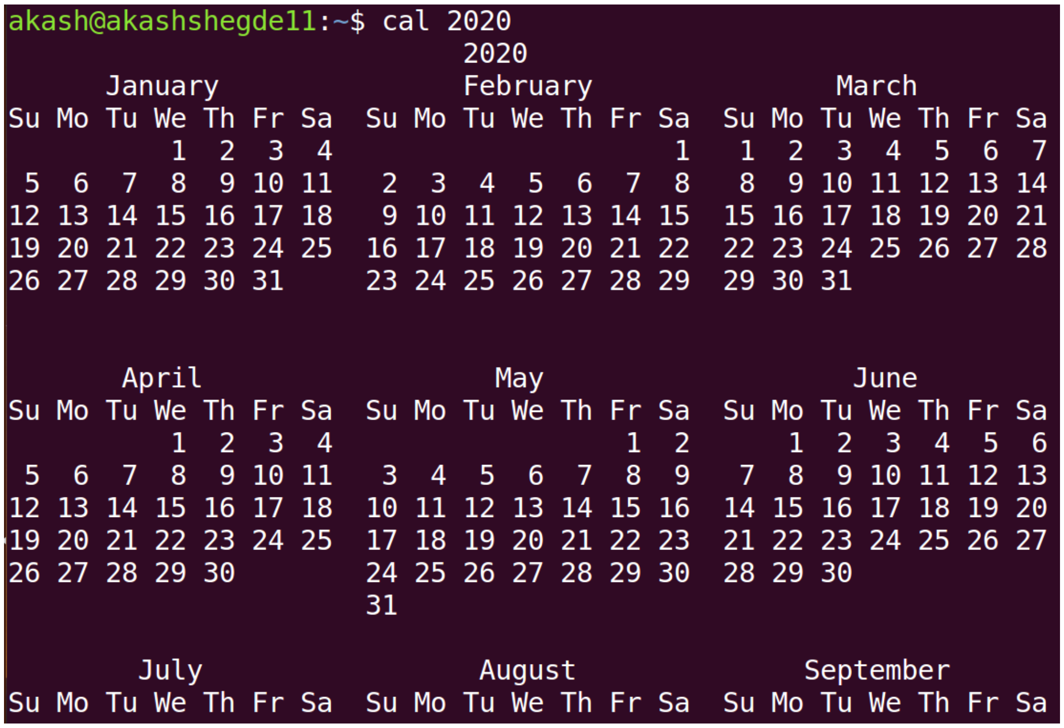
```
akash@akashshegde11:~$ cal
    October 2020
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

- The calendar of a specific month and year can be seen as below:



```
akash@akashshegde11:~$ cal 03 2011
     March 2011
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
```

- The calendar of a specific year also can be seen as below:

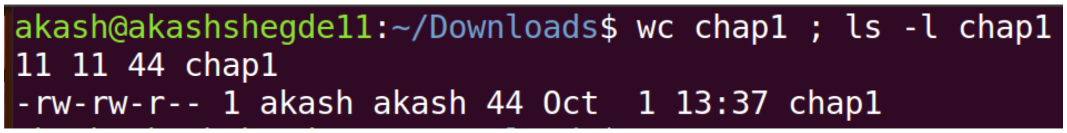


```
akash@akashshegde11:~$ cal 2020
                            2020
      January               February               March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4                     1   1  2  3  4  5  6  7
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   8  9 10 11 12 13 14
12 13 14 15 16 17 18   9 10 11 12 13 14 15  15 16 17 18 19 20 21
19 20 21 22 23 24 25  16 17 18 19 20 21 22  22 23 24 25 26 27 28
26 27 28 29 30 31     23 24 25 26 27 28 29  29 30 31


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

        July                 August              September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
```

The output will be seen across more than one page. It should be scrolled using the mouse or can be used with the ***less*** command to scroll with the keyboard.

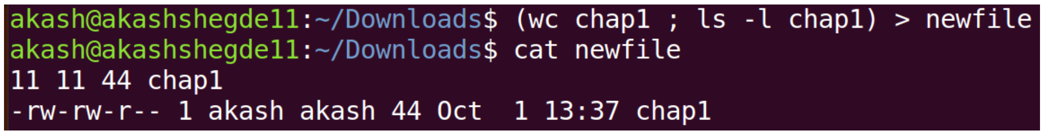
(IX) Combining Commands:

- The UNIX command line is flexible in allowing more than one command to be written and executed on the command line.
- Two or more commands to be executed are separated by a **;** (semicolon), which is used by the shell to understand the separate processing of these commands.
- An example with the combination of ***wc*** command and ***ls*** command is given below:



```
akash@akashshegde11:~/Downloads$ wc chap1 ; ls -l chap1
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

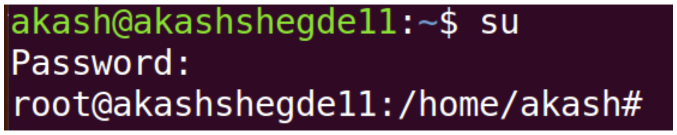
- The redirection of output from these combined commands can be done on the terminal as well.
- Suppose we wish to redirect the output obtained in the previous combination of ***wc*** and ***ls*** commands to a new file called *newfile*, it can be done as follows with > symbol:



```
akash@akashshegde11:~/Downloads$ (wc chap1 ; ls -l chap1) > newfile
akash@akashshegde11:~/Downloads$ cat newfile
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

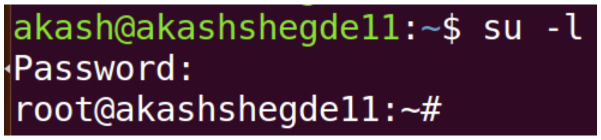
(X) Root Access:

- The *root* user or the *superuser* is the system administrator who has access to everything in the UNIX system.
- The root user performs all sorts of system administration tasks – maintaining user accounts, security, managing disk space to perform backups, and so on.
- The system administrator has this special login name, which is ***root***.
- This root user comes along with every UNIX system and the prompt for the root user is given by the # (hash) symbol.
- The superuser status can be acquired by using the ***su*** command as shown below:



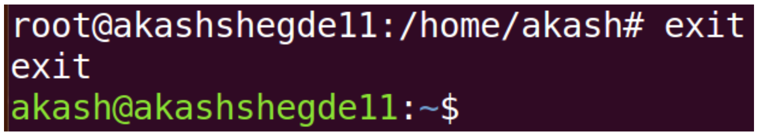
```
akash@akashshegde11:~$ su
Password:
root@akashshegde11:/home/akash#
```

- The prompt changes from **$** to **#** when this command is executed, but the directory remains the same.
  To move to the root's home directory upon acquiring superuser status, use ***su –l*** command as shown below:



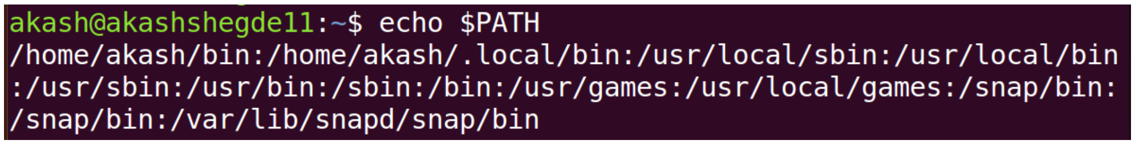
```
akash@akashshegde11:~$ su -l
Password:
root@akashshegde11:~#
```

- Leaving the root environment can be done by using ***exit*** command, as shown below:

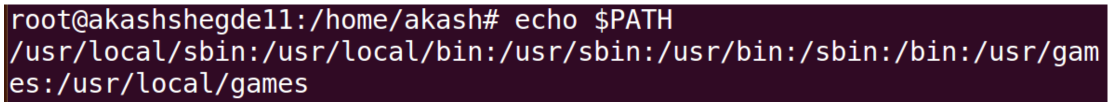


```
root@akashshegde11:/home/akash# exit
exit
akash@akashshegde11:~$
```

- The superuser constantly navigates the UNIX file system for administrative tasks, therefore the **PATH** variable is different for the superuser than other users and it does not include directory.
  The clear distinction between the **PATH** variables for other user and superuser can be seen below:



```
akash@akashshegde11:~$ echo $PATH
/home/akash/bin:/home/akash/.local/bin:/usr/local/sbin:/usr/local/bin
:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:
/snap/bin:/var/lib/snapd/snap/bin
```



```
root@akashshegde11:/home/akash# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/gam
es:/usr/local/games
```

- If any user *user1* is facing any issues with this system environment, the root user can recreate the same environment without the need of that user's password.
  This can be done with the command – ***su – user1***

(XI) UNIX Files – Introduction:

- A *file* in UNIX is a container for storing information.
- UNIX files do not contain the end-of-file (EOF) mark at the end of a file.
- All the file attributes of a UNIX file, such as name, size, last modified time, are all stored in the kernel and are not accessible by humans.
- Everything is a file in UNIX – commands, strings, characters, directories, devices, and so on.

(XII) File Types:

- There are three main types of UNIX files – ordinary files, directory files and device files.
- **Ordinary file** –
  - An ordinary file contains only data as a stream of characters.
  - It is also known as a *regular* file.
  - There are two types of ordinary files –
    - *Text file* –
      Text file contains only printable characters.
      Examples for text files include C, Java programs, shell, perl scripts.
      These files contain characters where every line is terminated with the *newline* character (*line feed*).
    - *Binary file* –
      Binary file contains both printable and unprintable characters from the full ASCII range.
      Examples for binary files include UNIX commands, object code and executables, pictures, audio and video files.
      These files usually cannot be displayed properly on the screen.
- **Directory file** –
  - A directory file contains names of files and other directories, and a number associated with each name.
  - It contains no data, but has details of all files and subdirectories under it.
  - It contains an entry for every file and subdirectory in that particular directory.
  - Each entry has two components, namely *filename* and a unique identification number called the *inode* number.
  - A directory contains only the filenames and not the contents of the files themselves.
  - Users cannot write to a directory, but performing actions like creating or removing files updates the contents of the directory.

- **Device file** –
  - All devices and peripherals attached to the computer system are represented by device files.
  - It does not contain any information at all.
  - The operation of a device is governed by attributes of its associated file.
  - The kernel identifies a device from the attributes of its associated file and uses these attributes to operate the device.
    For example, copying files to a pendrive or a micro-SD card.
  - Device filenames are usually located at **/dev**

(XIII) Naming Files:

- A filename in UNIX can consist of up to 255 characters and is case-sensitive.
- File names may or may not have extensions in UNIX and can contain any character except / and **NULL** character (ASCII value 0).
- Filenames can have control characters or other unprintable characters in them.
- Some recommended characters for UNIX filenames are:
  - Alphabetic characters and numerals.
  - **.** (dot), **-** (hyphen) and **_** (underscore).
- **-** (hyphen) is not recommended at the beginning of a filename as it interferes with command options.

(XIV) Organization of Files:

- The UNIX file system is organized as an inverted tree structure, as shown below.

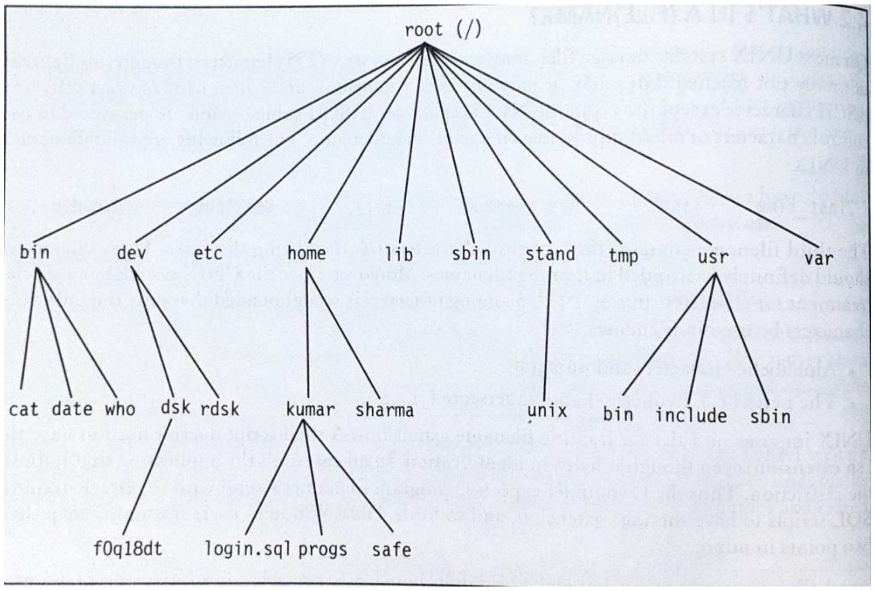


Figure 2. The UNIX file system tree: The Parent-Child Relationship

- All files in UNIX are related to one another, with the topmost file being the root directory file, (represented by /). It is not the same as the root user.
- The UNIX file system is a complete collection of all files in a hierarchical manner.
- The root is followed by several directories, which further have several subdirectories, and so on.
- Every file in UNIX has a parent, apart from root.

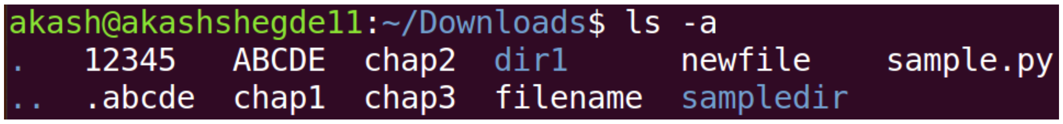
(XV) Standard Directories:

- There are a few standard directories in UNIX and these are given below:

Directory	**Significance**
/bin	Programs needed for using and managing the system (binaries)
/dev	System device files – interface to a particular device
/etc	System-specific configuration files and files essential for start-up
/home	Home directories for all users of the system
/mnt	Temporary file systems are mounted
/opt	Software files that are not installed when the OS is installed (products by third-party vendors)
/sbin	Programs for system administration (system binaries)
/tmp	Holding temporary files (scratch directory)
/usr	Programs and data related to users of a system (read-only and can be shared on a network)
/var	Files with varying content (log files, mail system files, print spooling system files)

(XVI) Hidden Files:

- Files that are not usually displayed in a directory listing are known as *hidden* files.
- The filenames for such files begin with a **.** (dot).
- These files only show up when the full listing is used, such as ***ls –a*** command.



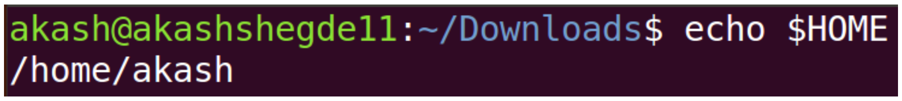
```
akash@akashshegde11:~/Downloads$ ls -a
.   12345   ABCDE  chap2  dir1      newfile   sample.py
..  .abcde  chap1  chap3  filename  sampledir
```

Here, ***.abcde*** is a hidden file.

- The concept of hidden files came into existence in order to store configuration details and informational text.
- These files are also known as *dotfiles*.

(XVII) Home Directory and HOME Variable:

- The default directory upon logging in to the UNIX system is the *home* directory.
- It is created by the system when a user account is opened.
- The name used for login gives the user a separate directory under the home directory.
- The shell environment variable **HOME** knows the location of the home directory, and is invoked using the following command:

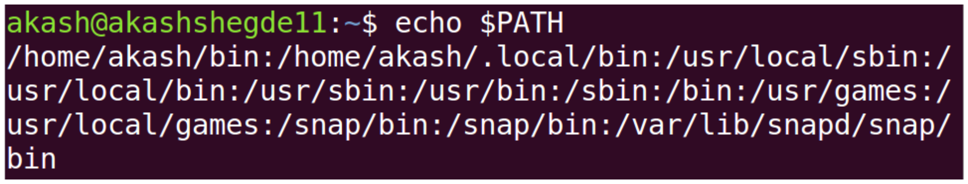


```
akash@akashshegde11:~/Downloads$ echo $HOME
/home/akash
```

- The ~ (tilde) symbol followed by the / (forward slash) is used to refer to the home directory.

(XVIII) PATH Variable:

- The **PATH** environment variable specifies a set of directories where programs are stored.
- It can be seen using the following command:



```
akash@akashshegde11:~$ echo $PATH
/home/akash/bin:/home/akash/.local/bin:/usr/local/sbin:/
usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
usr/local/games:/snap/bin:/snap/bin:/var/lib/snapd/snap/
bin
```

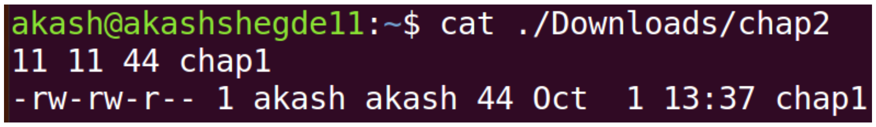
- The **PATH** can also be manipulated in two ways, namely –
  - by changing the value of **PATH** to include a specific directory.
  - by using a pathname in the command line.

(XIX) Absolute and Relative Pathnames:

- **Absolute Pathname** –
  - The absolute pathname refers to the location of a file with respect to the root directory /
  - It always starts with a /
  - Example: **/home/abcde/sample.txt** is the absolute pathname of a file *sample.txt* from the root directory.
- **Relative Pathname** –
  - The relative pathname refers to the location of a file with respect to the current working directory.
  - It never starts with a /
  - Example (if current working directory is **/home**): **abcde/sample.txt** is the relative pathname of a file *sample.txt* from its current working directory.

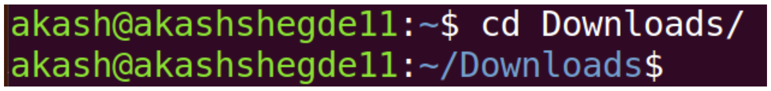
(XX) Dot and Double Dot Notations:

- **. (single dot) notation** –
  - It refers to the current directory.
  - An example with the cat command is shown below:

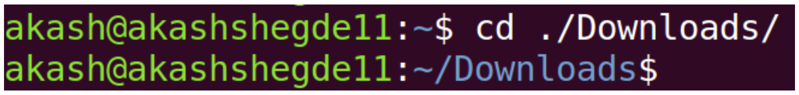


```
akash@akashshegde11:~$ cat ./Downloads/chap2
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

  - The **./** can also be specified when referring to a file/directory using the relative pathname.

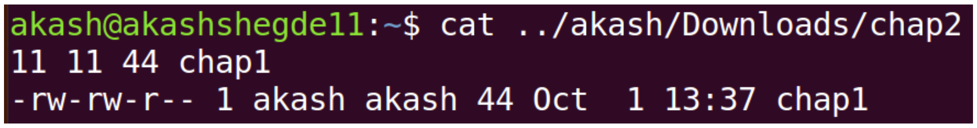


```
akash@akashshegde11:~$ cd Downloads/
akash@akashshegde11:~/Downloads$
```



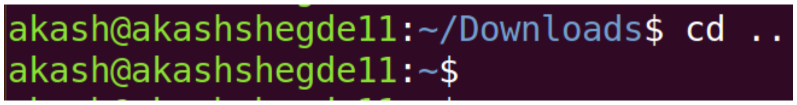
```
akash@akashshegde11:~$ cd ./Downloads/
akash@akashshegde11:~/Downloads$
```

- **.. (double dot notation)** –
  - It refers to the parent directory of the current directory.
  - An example with the cat command is shown below:



```
akash@akashshegde11:~$ cat ../akash/Downloads/chap2
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

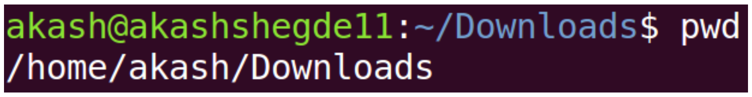
  - The **..** can be used with cd command to move up one level in the file system.



```
akash@akashshegde11:~/Downloads$ cd ..
akash@akashshegde11:~$
```

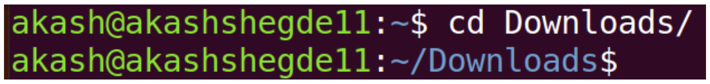
(XXI) Directory Commands:

- There are several commands that are used to work with directories.
- ***pwd*** command –
  pwd command is used to print the current working directory.
- It gives the absolute path of the current directory where the user is located.
  An example of the ***pwd*** command is shown below:



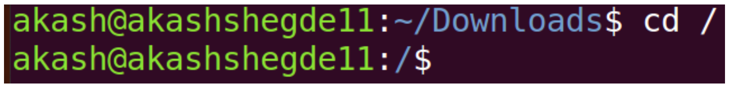
```
akash@akashshegde11:~/Downloads$ pwd
/home/akash/Downloads
```

- ***cd*** command –
  cd command is used to change the current working directory.



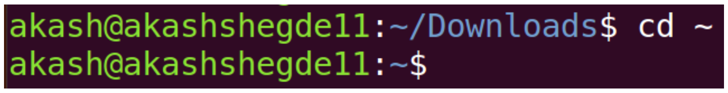
```
akash@akashshegde11:~$ cd Downloads/
akash@akashshegde11:~/Downloads$
```

- Changing to *root* directory can be done with ***cd /***



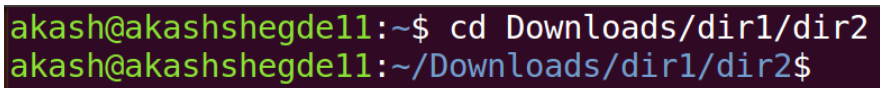
```
akash@akashshegde11:~/Downloads$ cd /
akash@akashshegde11:/$
```

- Changing to *home* directory can be done either with ***cd ~*** or just ***cd***



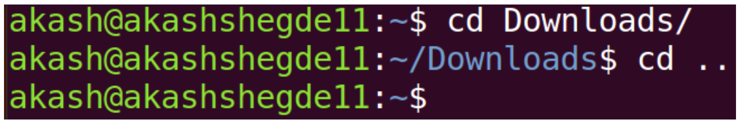
```
akash@akashshegde11:~/Downloads$ cd ~
akash@akashshegde11:~$
```

- Changing to some directory can be done with ***cd dir1/dir2/dir3***



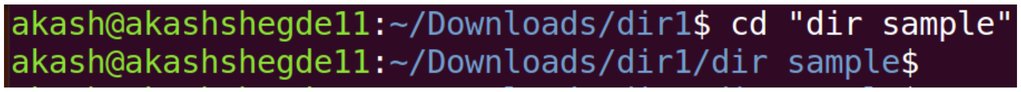
```
akash@akashshegde11:~$ cd Downloads/dir1/dir2
akash@akashshegde11:~/Downloads/dir1/dir2$
```

- Changing to the parent directory of the current directory can be done with ***cd ..***



```
akash@akashshegde11:~$ cd Downloads/
akash@akashshegde11:~/Downloads$ cd ..
akash@akashshegde11:~$
```

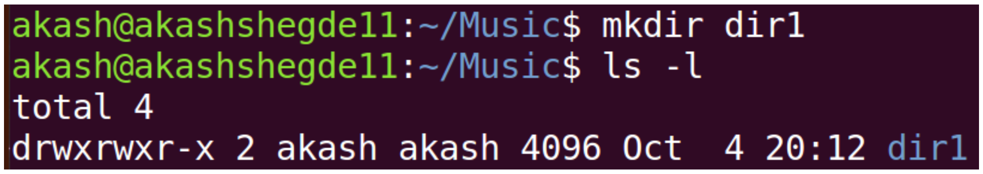
- Changing to a directory which has spaces in its name can be done either with ***cd "dir name"*** or ***cd dir\ name***



```
akash@akashshegde11:~/Downloads/dir1$ cd "dir sample"
akash@akashshegde11:~/Downloads/dir1/dir sample$
```

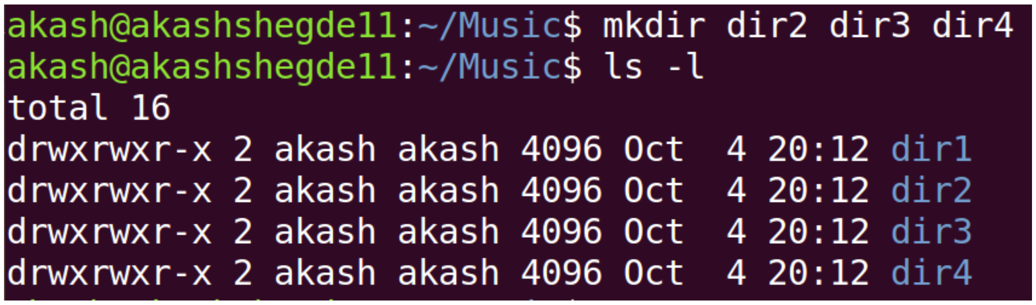
- ***cd*** command does not require the absolute path when specifying the argument.

- ***mkdir*** command –
  mkdir command is used to create a new directory.
- One directory can be created with ***mkdir dir1***



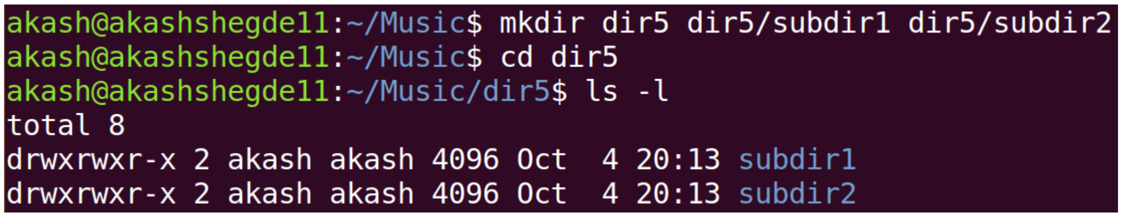
```
akash@akashshegde11:~/Music$ mkdir dir1
akash@akashshegde11:~/Music$ ls -l
total 4
drwxrwxr-x 2 akash akash 4096 Oct  4 20:12 dir1
```

- Multiple directories can be created with ***mkdir dir1 dir2 dir3***



```
akash@akashshegde11:~/Music$ mkdir dir2 dir3 dir4
akash@akashshegde11:~/Music$ ls -l
total 16
drwxrwxr-x 2 akash akash 4096 Oct  4 20:12 dir1
drwxrwxr-x 2 akash akash 4096 Oct  4 20:12 dir2
drwxrwxr-x 2 akash akash 4096 Oct  4 20:12 dir3
drwxrwxr-x 2 akash akash 4096 Oct  4 20:12 dir4
```

- Directory trees can be created with ***mkdir dir1 dir1/subdir1 dir1/subdir2***

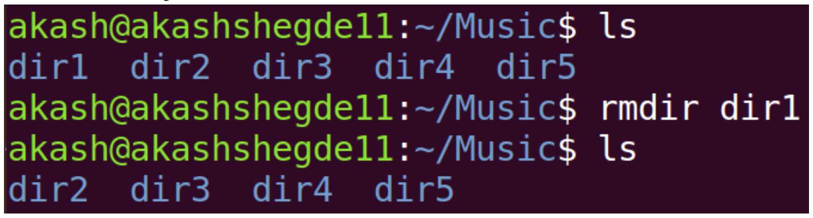


```
akash@akashshegde11:~/Music$ mkdir dir5 dir5/subdir1 dir5/subdir2
akash@akashshegde11:~/Music$ cd dir5
akash@akashshegde11:~/Music/dir5$ ls -l
total 8
drwxrwxr-x 2 akash akash 4096 Oct  4 20:13 subdir1
drwxrwxr-x 2 akash akash 4096 Oct  4 20:13 subdir2
```

- The order of the arguments is extremely important when creating directory trees.

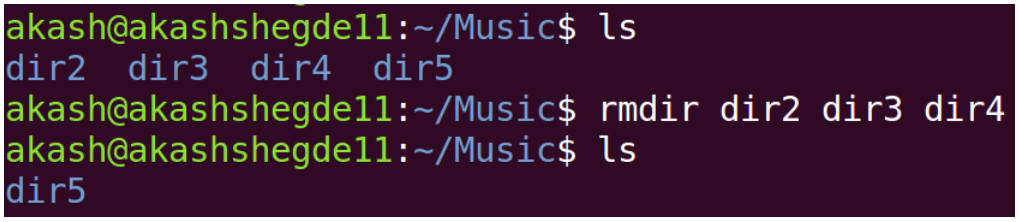
- Sometimes, directory creation may fail. This could be due to one or more reasons, namely –
    - Directory already exists.
    - Ordinary file exists with the same name.
    - Insufficient permissions to create a directory.

- ***rmdir*** command –
  rmdir command is used to remove a directory.
- The directory to be removed has to be empty, otherwise it cannot be removed.
- However, the contents inside it may get removed silently.
- One directory can be removed with ***rmdir dir1***



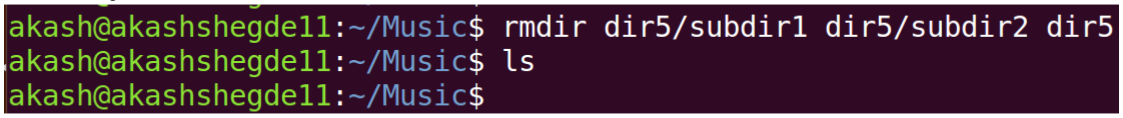
```
akash@akashshegde11:~/Music$ ls
dir1  dir2  dir3  dir4  dir5
akash@akashshegde11:~/Music$ rmdir dir1
akash@akashshegde11:~/Music$ ls
dir2  dir3  dir4  dir5
```

- Multiple directories can be removed with ***rmdir dir1 dir2 dir3***



```
akash@akashshegde11:~/Music$ ls
dir2  dir3  dir4  dir5
akash@akashshegde11:~/Music$ rmdir dir2 dir3 dir4
akash@akashshegde11:~/Music$ ls
dir5
```

- Directory trees can be removed with ***rmdir dir1/subdir1 dir1/subdir2 dir1***

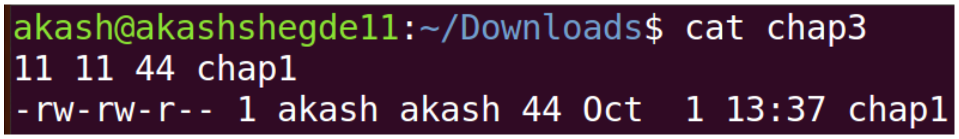


```
akash@akashshegde11:~/Music$ rmdir dir5/subdir1 dir5/subdir2 dir5
akash@akashshegde11:~/Music$ ls
akash@akashshegde11:~/Music$
```

- The reverse order of the arguments is extremely important when removing directory trees.
- Sometimes, directory removal may fail. This could be due to one or more reasons, namely –
    - Directory is not empty.
    - User is not present in the parent directory of the directory to be removed.
    - Insufficient permissions to remove the directory.

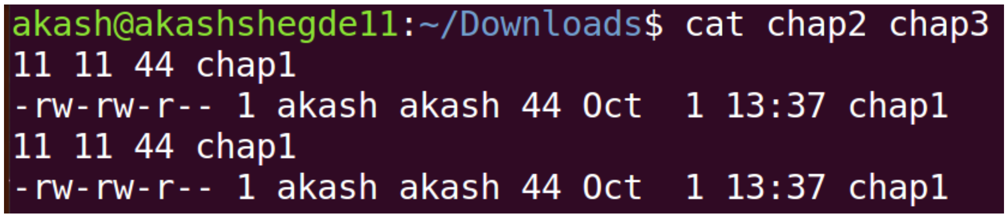
(XXII) File-related Commands:

- There are several commands that are used to work with files.
- ***cat*** command –
  cat command is used to display the contents of a file on the terminal.



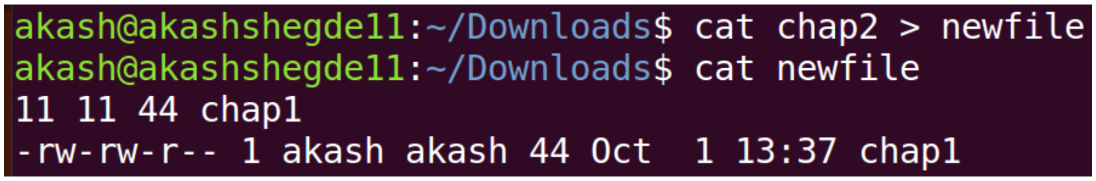
```
akash@akashshegde11:~/Downloads$ cat chap3
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

- The ***cat*** command also accepts multiple filenames as arguments.
  It just prints the concatenated output to the terminal.



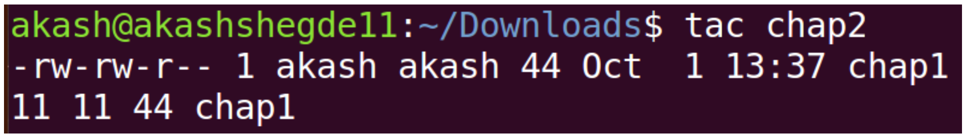
```
akash@akashshegde11:~/Downloads$ cat chap2 chap3
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

- There are two additional options that can be used with the ***cat*** command, namely –
  - ***-v*** option – display non-printable characters.
  - ***-n*** option – numbering the lines in the output.
- A new file can be created using the ***cat*** command, as follows:



```
akash@akashshegde11:~/Downloads$ cat chap2 > newfile
akash@akashshegde11:~/Downloads$ cat newfile
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

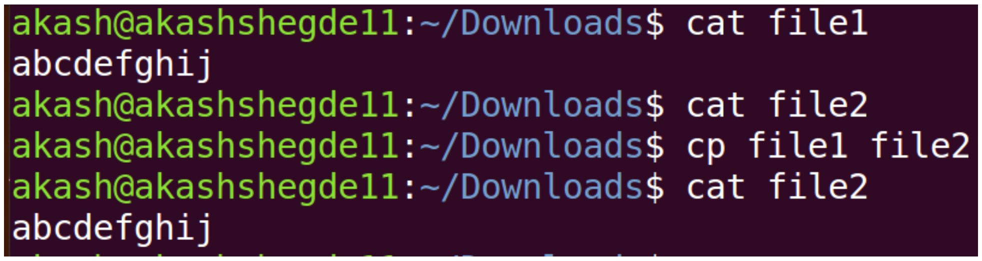
- A file can be printed in reverse using the ***tac*** command, which is the reverse of the ***cat*** command.



```
akash@akashshegde11:~/Downloads$ tac chap2
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
11 11 44 chap1
```

- As it can be seen, the ***cat*** command is very versatile in creating, displaying, concatenating and appending content to files.

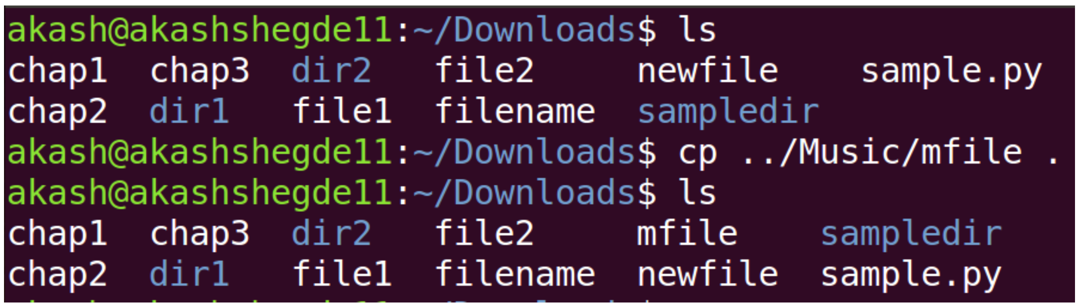
- ***cp*** command –
  cp command is used to copy a file or a group of files.
- It requires at least two filenames to be specified in the command line.
  An example is shown below:



```
akash@akashshegde11:~/Downloads$ cat file1
abcdefghij
akash@akashshegde11:~/Downloads$ cat file2
akash@akashshegde11:~/Downloads$ cp file1 file2
akash@akashshegde11:~/Downloads$ cat file2
abcdefghij
```

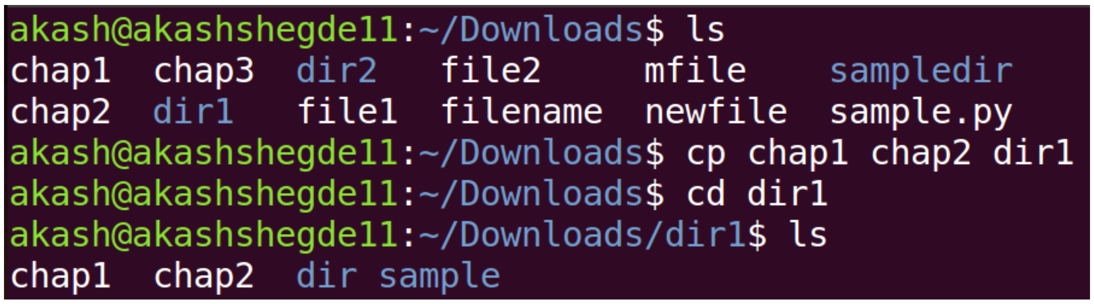
- If the destination file does not exist, then the ***cp*** command creates the file before copying the contents to it.
- If the destination file already exists, then the ***cp*** command overwrites it without any warning to the user.
- If there is only one file to be copied, the destination can be either an ordinary file or a directory.

- The ***cp*** command is often used to copy files to the current directory, as shown below:



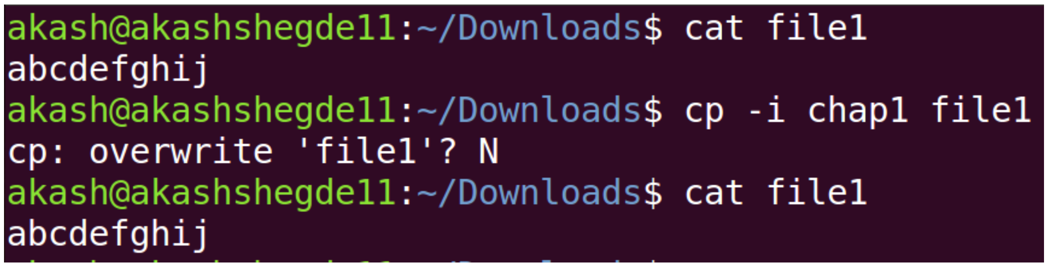
```
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2   file2     newfile   sample.py
chap2  dir1   file1  filename  sampledir
akash@akashshegde11:~/Downloads$ cp ../Music/mfile .
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2   file2     mfile    sampledir
chap2  dir1   file1  filename  newfile  sample.py
```

- If there are multiple files to be copied, the last filename must be a directory and that directory must exist.
  An example is shown below:



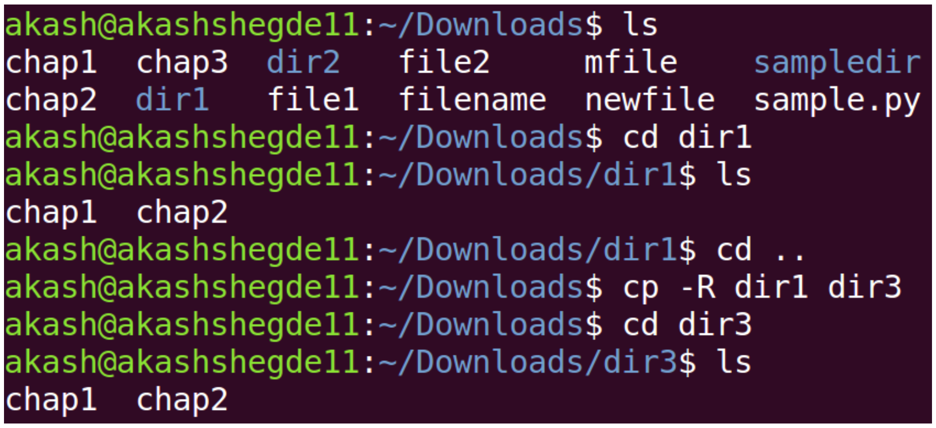
```
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2   file2     mfile    sampledir
chap2  dir1   file1  filename  newfile  sample.py
akash@akashshegde11:~/Downloads$ cp chap1 chap2 dir1
akash@akashshegde11:~/Downloads$ cd dir1
akash@akashshegde11:~/Downloads/dir1$ ls
chap1  chap2  dir sample
```

- The ***cp*** command offers an interactive copying mechanism with the ***–i*** option, which warns the user before overwriting a file.
  An example is shown below:



```
akash@akashshegde11:~/Downloads$ cat file1
abcdefghij
akash@akashshegde11:~/Downloads$ cp -i chap1 file1
cp: overwrite 'file1'? N
akash@akashshegde11:~/Downloads$ cat file1
abcdefghij
```

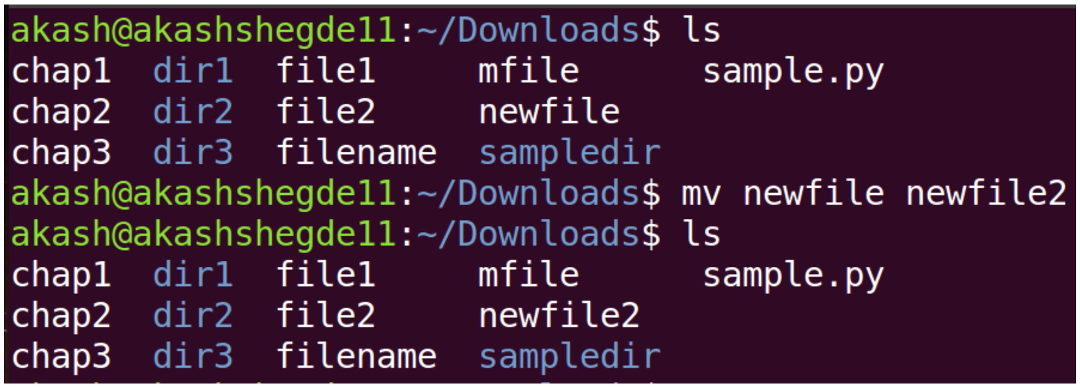
- The ***cp*** command also allows for recursive copying with the ***–R*** option, where it copies an entire directory structure and all the subdirectories.



```
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2   file2     mfile    sampledir
chap2  dir1   file1  filename  newfile  sample.py
akash@akashshegde11:~/Downloads$ cd dir1
akash@akashshegde11:~/Downloads/dir1$ ls
chap1  chap2
akash@akashshegde11:~/Downloads/dir1$ cd ..
akash@akashshegde11:~/Downloads$ cp -R dir1 dir3
akash@akashshegde11:~/Downloads$ cd dir3
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2
```

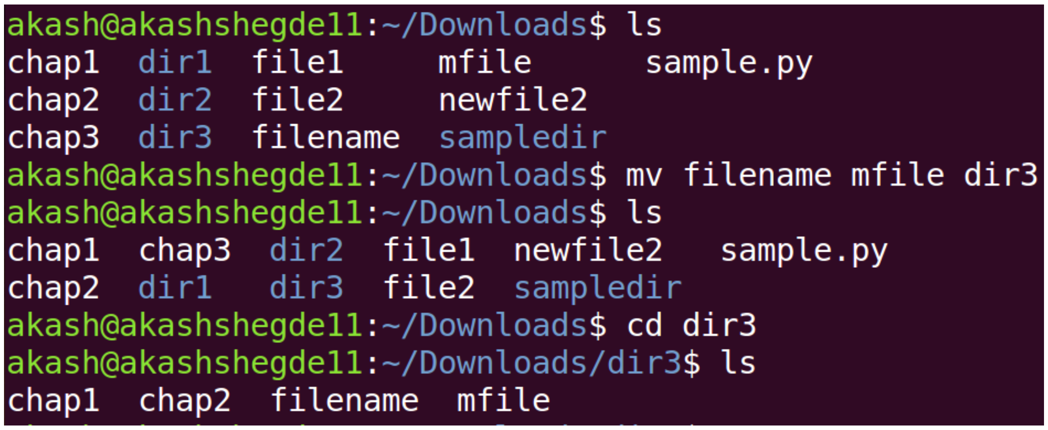
- If it is not possible to copy a file, it would either be a read-protected file or the destination would be write-protected.

- ***mv*** command –
  mv command is used to move files.
- This command has two distinct functions, namely renaming a file and moving multiple files to a different directory.
- An example is shown below:



```
akash@akashshegde11:~/Downloads$ ls
chap1  dir1  file1     mfile      sample.py
chap2  dir2  file2     newfile
chap3  dir3  filename  sampledir
akash@akashshegde11:~/Downloads$ mv newfile newfile2
akash@akashshegde11:~/Downloads$ ls
chap1  dir1  file1     mfile      sample.py
chap2  dir2  file2     newfile2
chap3  dir3  filename  sampledir
```

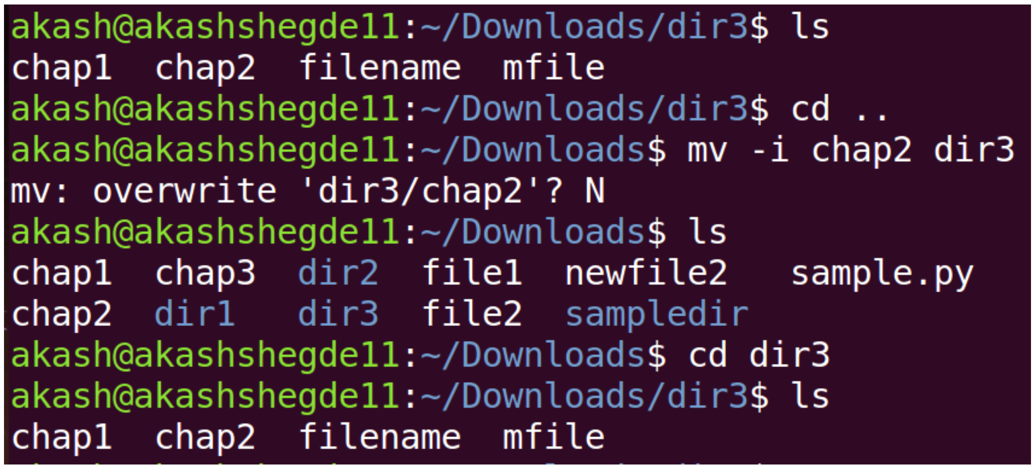
- If the destination file does not exist, then the ***mv*** command creates the file before moving the contents to it.
- If the destination file already exists, then the ***mv*** command overwrites it without any warning to the user.
- If there are multiple files to be moved, the last filename must be a directory and that directory must exist.
  An example is shown below:



```
akash@akashshegde11:~/Downloads$ ls
chap1  dir1  file1     mfile      sample.py
chap2  dir2  file2     newfile2
chap3  dir3  filename  sampledir
akash@akashshegde11:~/Downloads$ mv filename mfile dir3
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2  file1  newfile2   sample.py
chap2  dir1   dir3  file2  sampledir
akash@akashshegde11:~/Downloads$ cd dir3
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename  mfile
```

- The ***mv*** command offers an interactive moving mechanism with the ***–i*** option, which warns the user before overwriting a file.

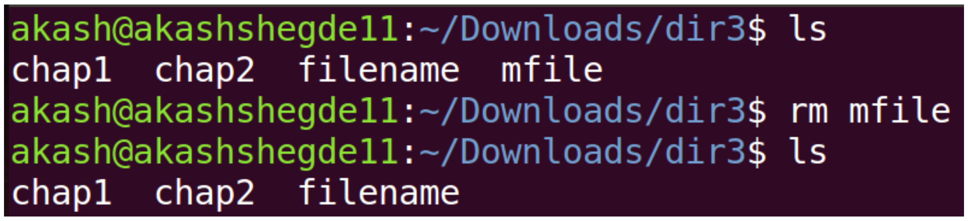
An example is shown below:



```
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename  mfile
akash@akashshegde11:~/Downloads/dir3$ cd ..
akash@akashshegde11:~/Downloads$ mv -i chap2 dir3
mv: overwrite 'dir3/chap2'? N
akash@akashshegde11:~/Downloads$ ls
chap1  chap3  dir2  file1  newfile2   sample.py
chap2  dir1   dir3  file2  sampledir
akash@akashshegde11:~/Downloads$ cd dir3
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename  mfile
```

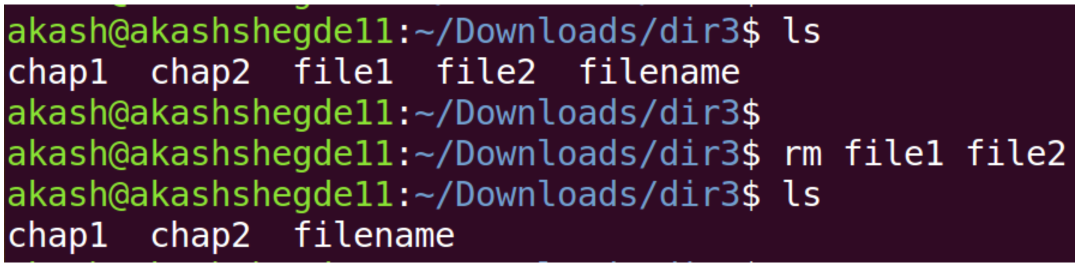
- If it is not possible to move a file, it would either be a read-protected file or the destination would be write-protected.

- ***rm*** command –
  rm command is used to remove/delete one or more files.
- It operates silently and should be used with caution.
- A file once deleted using the ***rm*** command cannot be recovered.
  An example is shown below:



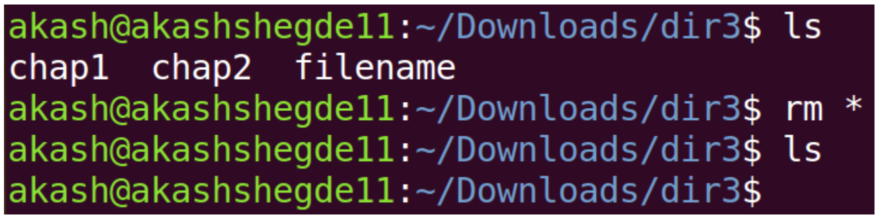
```
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename  mfile
akash@akashshegde11:~/Downloads/dir3$ rm mfile
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename
```

- Multiple files can be deleted at once using the ***rm*** command, as shown below.



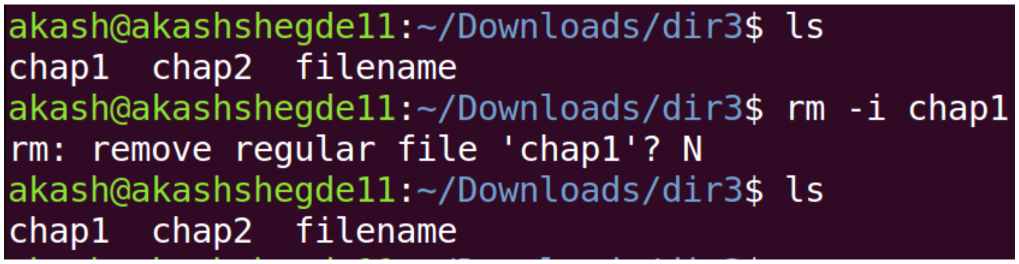
```
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  file1  file2  filename
akash@akashshegde11:~/Downloads/dir3$
akash@akashshegde11:~/Downloads/dir3$ rm file1 file2
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename
```

- The ***rm*** command won't normally remove a directory, but it can remove files that are present within a directory.
- All files within a directory can be removed using the following command:



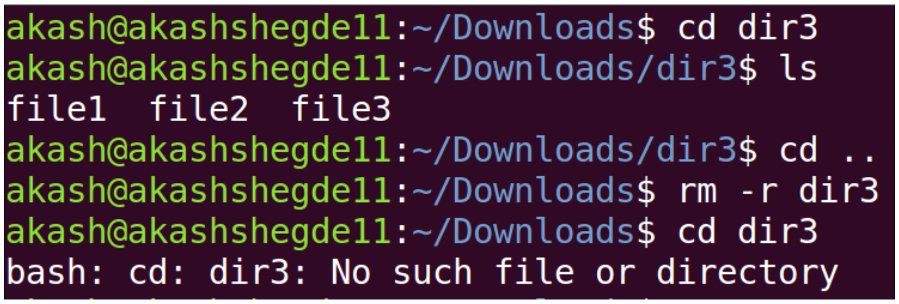
```
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename
akash@akashshegde11:~/Downloads/dir3$ rm *
akash@akashshegde11:~/Downloads/dir3$ ls
akash@akashshegde11:~/Downloads/dir3$
```

- The ***rm*** command offers an interactive deletion mechanism with the ***–i*** option, which warns the user before deleting a file.
  An example is shown below:



```
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename
akash@akashshegde11:~/Downloads/dir3$ rm -i chap1
rm: remove regular file 'chap1'? N
akash@akashshegde11:~/Downloads/dir3$ ls
chap1  chap2  filename
```

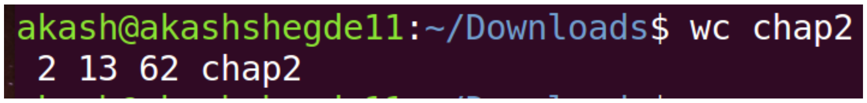
- The ***rm*** command also allows for recursive deletion with the ***–r*** or ***–R*** option, where it deletes an entire directory structure and all the subdirectories and files recursively.



```
akash@akashshegde11:~/Downloads$ cd dir3
akash@akashshegde11:~/Downloads/dir3$ ls
file1  file2  file3
akash@akashshegde11:~/Downloads/dir3$ cd ..
akash@akashshegde11:~/Downloads$ rm -r dir3
akash@akashshegde11:~/Downloads$ cd dir3
bash: cd: dir3: No such file or directory
```

- The recursive deletion thoroughly scans through the entire directory tree and then deletes everything.
- If there are any write-protected files that have to be removed, the ***rm*** command can be used with the ***–f*** option to delete such files.
- One of the most dangerous commands in UNIX is a combination of the above options, yielding ***rm –rf ****
  This combination deletes everything in a particular directory without warning the user.
  If this command is used in the root directory, it will wipe the entire UNIX system!

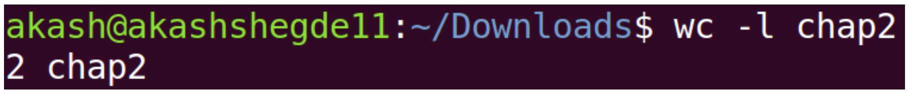
- ***wc*** command –
  wc command counts and displays the number of lines, number of words and number of characters in a file or a stream of data.
- It takes in one or more filenames as arguments and displays a four-columnar output, which gives the number of lines, number of words, number of characters in a file and the filename respectively.
  An example is shown below:



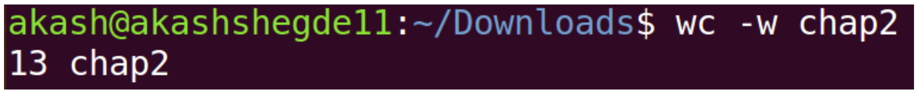
```
akash@akashshegde11:~/Downloads$ wc chap2
 2 13 62 chap2
```

- If the user wants a specific output for the number of lines, number of words or number of characters in a file, the ***–l***, ***–w*** and ***–c*** options can be used respectively.

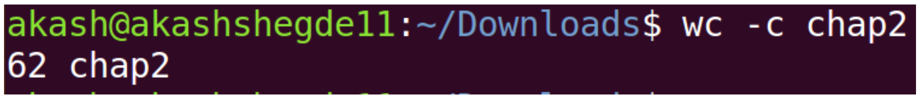
An example is shown below:



```
akash@akashshegde11:~/Downloads$ wc -l chap2
2 chap2
```



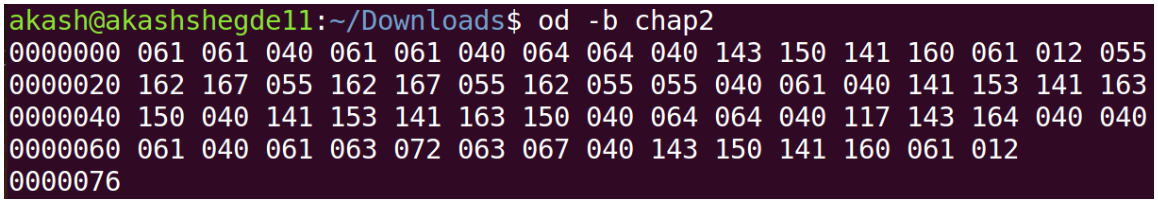
```
akash@akashshegde11:~/Downloads$ wc -w chap2
13 chap2
```



```
akash@akashshegde11:~/Downloads$ wc -c chap2
62 chap2
```

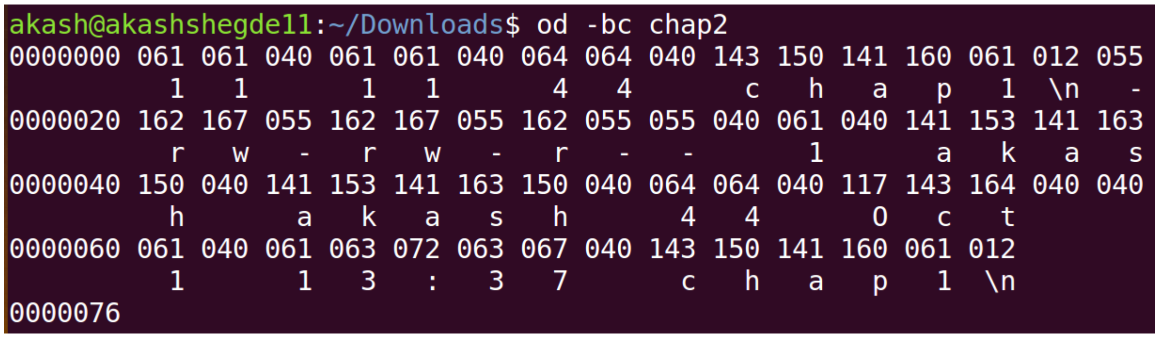
- If multiple filenames are provided as arguments, the total count of the lines, words and characters of all the files will also be shown at the end.

- ***od*** command –
  od command displays the octal dump of the specified data of a file.
- It requires an option and a filename to display readable output.
- The ***od*** command uses ***–b*** option to display octal values of each character separately.
  An example is shown below:



```
akash@akashshegde11:~/Downloads$ od -b chap2
0000000 061 061 040 061 061 040 064 064 040 143 150 141 160 061 012 055
0000020 162 167 055 162 167 055 162 055 055 040 061 040 141 153 141 163
0000040 150 040 141 153 141 163 150 040 064 064 040 117 143 164 040 040
0000060 061 040 061 063 072 063 067 040 143 150 141 160 061 012
0000076
```

- The ***od*** command uses ***–c*** option to display the characters. However, it can be combined with ***–b*** option as ***–bc*** option, which gives the characters as well as their octal values, thereby providing the most clear output to the user.



```
akash@akashshegde11:~/Downloads$ od -bc chap2
0000000 061 061 040 061 061 040 064 064 040 143 150 141 160 061 012 055
          1   1       1   1       4   4       c   h   a   p   1  \n   -
0000020 162 167 055 162 167 055 162 055 055 040 061 040 141 153 141 163
          r   w   -   r   w   -   r   -   -       1       a   k   a   s
0000040 150 040 141 153 141 163 150 040 064 064 040 117 143 164 040 040
          h       a   k   a   s   h       4   4       O   c   t
0000060 061 040 061 063 072 063 067 040 143 150 141 160 061 012
          1       1   3   :   3   7       c   h   a   p   1  \n
0000076
```

- The octal representations of the characters are given in the first line of the output, whereas the printable characters and escape sequences are given in the second line.