Linux Notes

Dr. J. V. Smart

Table of Contents

- Syllabus
- Glossary
- Introduction
- The Unix/Linux File System
  - Accessing Multiple File Systems
- The Shell
  - The Linux Shells
- The Bourne Shell
  - The Command Line
- File System Manipulation Commands
- Shell Globbing Patterns
- Plain Text Editors
  - The nano Editor
- The Shell Scripting Language of the *bash* Shell
  - Shell Variables
  - The `test` Command and the `[` Builtin
  - The Control Structures
  - Comments
  - Output
  - Input
  - Integer Arithmetic
- Commands List

Syllabus

Syllabus with effect from the Academic Year 2021-2022

Course Code	PS01CMCA54	Title of the Course	OPERATING SYSTEMS
Total Credits of the Course	4	Hours per Week	4

Course Objectives:	1. To provide basic understanding of the role and functioning of an operating system. 2. To introduce Linux shell environment and programming.

Course Content

Unit	Description	Weightage* (%)
1.	**Introduction to Operating Systems** - Understanding the role of operating systems - Operating system services - Operating system structure - The concepts of interrupt handling, system call, shell, operating system interface - Virtual machines - Linux Bash shell programming fundamentals - Command-line processing - Bash shell variables, control structures - input, output, integer arithmetic, string operations	25
2.	**Process Management** - The concept of a process - Scheduling of processes - Interprocess communication - Multithreading: concepts, advantages, models - Schedulers: long term, middle term, short term - CPU scheduling: criteria and algorithms - Multiprocessor scheduling - Introduction to process synchronization - The critical section problem and Peterson's solution - The concepts of semaphores and monitors	25

Unit	Description	Weightage* (%)
	- Introduction to deadlocks	
3.	**Memory Management and File Systems** - Basic concepts of memory management - Paging - Segmentation - Virtual memory, demand paging - Page replacement - Introduction to file system management and directory structure - File system mounting - Disk scheduling	25
4.	***Linux Shell Programming* - The vim editor - File system manipulation commands - I/O redirection - Regular expressions - Basic filters - The sed and awk commands	25

Teaching-Learning Methodology	Blended learning approach incorporating traditional classroom teaching as well as online / ICT-based teaching practices

Evaluation Pattern

Sr. No.	Details of the Evaluation	Weightage
1.	Internal Written / Practical Examination (As per CBCS R.6.8.3)	15%
2.	Internal Continuous Assessment in the form of Practical, Viva-voce, Quizzes, Seminars, Assignments, Attendance (As per CBCS R.6.8.3)	15%
3.	University Examination	70%

Course Outcomes: Having completed this course, the learner will be able to

Course Outcomes: Having completed this course, the learner will be able to	
1.	describe the role and functioning of an operating system.
2.	demonstrate understanding of fundamental concepts related to operating systems.
3.	understand process, memory and file system management.
4.	gain familiarity with Linux command line environment.
5.	use basic Linux commands.
6.	develop Linux shell scripts.

Suggested References:

Sr. No.	References
1.	Silbetschatz, Galvin, Gagne: Operating System Concepts, 8th edition, John Wiley and Sons, Inc., 2008
2.	Kochan S. G., Wood, P. : Unix Shell Programming, 4th edition, Addison Wesley, 2016
3.	Das S. : UNIX and Shell Programming, Tata McGraw-Hill Education, 2008
4.	Nutt G. : “Operating Systems” : 3rd Edition, Pearson Education, 2004
5.	Tanenbaum A. S., Woodhull A.S. : “Operating Systems Design and Implementation”, 3rd edition, Prentice Hall, 2006
6.	Shotts W. : “The Linux Command Line: A Complete Introduction Illustrated Edition”, 2nd Edition, No Starch Press, 2019

Glossary

Kernel

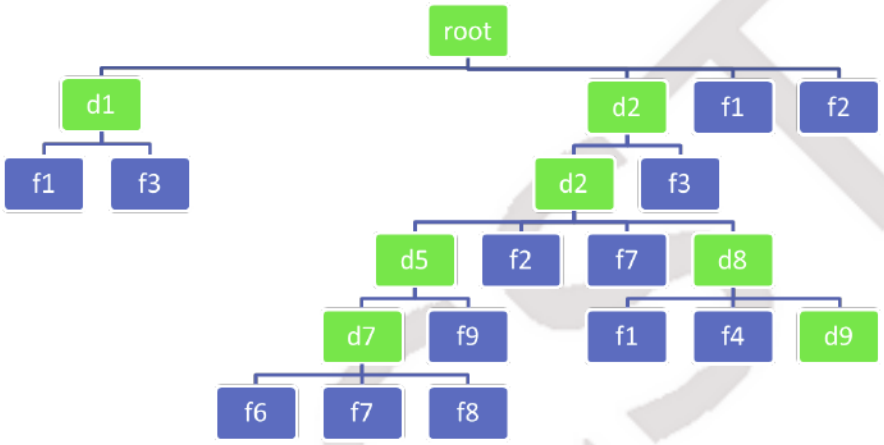
The operating system kernel is the one program running on the computer system at all times. The kernel provides the interface between the hardware and the programs

Shell

A shell is an interface between the user and the operating system. A shell provides either a Graphical User Interface (GUI) or a Command Line Interface (CLI)

Introduction

The Unix/Linux File System



The file system structure

- Hierarchical file system
  - While the Windows file systems is actually a forest, the UNIX file systems is a proper tree (actually a directed acyclic graph)
- The root directory is identified by the / (slash) character
- Directories and files
- Identifying a file system object uniquely using path
- Components of the path are separated by the / (forward slash) character
- / (forward slash) v/s \ (backslash)
- Filenames are case sensitive
- Extensions not an essential part of file name (executable files usually have no extension)
- Any character except / can be used in file names
- The notion of current directory and relative paths
- Absolute path v/s relative path
- Interactive users have a home directory. A user has full permissions on one's home directory. The ~ (tilde sign) is a shortcut for the user's home

directory in file system manipulation commands

- Hidden files and directories - any file or directory whose name starts with a . is considered hidden. Hidden files and directories are not displayed in the nautilus file browser (GUI) or by the ls command (CUI). However, there are options to display them

Accessing Multiple File Systems

A computer system may have many storage devices in it. Also, removable devices may be inserted or attached and removed at any time. Each device has its own file system on it. A device that can have multiple partitions, like the hard disk, has a separate file system on each partition. How does one access these file systems? Operating systems like Microsoft Windows assign a separate drive letter (like C:, D:, E:, etc.) to each file system. However, Linux and other Unix-like systems have a single file system tree starting with the root directory, denoted by / (the slash character). The file system contained on the partition from which Ubuntu boots is called the root file system. The root directory of this file system becomes / - the root of the entire file system tree. Initially this is the only file system available.

We may access any other file system by mounting it on any existing directory (this directory is called the mount point). Once mounted, the contents of that file system appear as the contents of the mount point directory. If the mount point previously contained some contents (files and subdirectories), they are masked (hidden) for the duration of the mount. Now we may access (and modify) the contents of that file system from the mount point directory. When we no longer need to use the file system, we may unmount it. At this point, the original contents of the mount point directory get unmasked (become visible again). This process is depicted in the following figures a, b, c and d. Figure a shows the root file system. Figure b shows the file system on another device. Figure c shows the situation after mounting the file system of figure b onto the directory d3 of Figure a. The original contents of d3 are now masked and the contents of the file system mounted there appear as if they are the contents of d3. Figure d shows the situation after unmounting the second file system. The original contents of the directory d3 now become visible again.

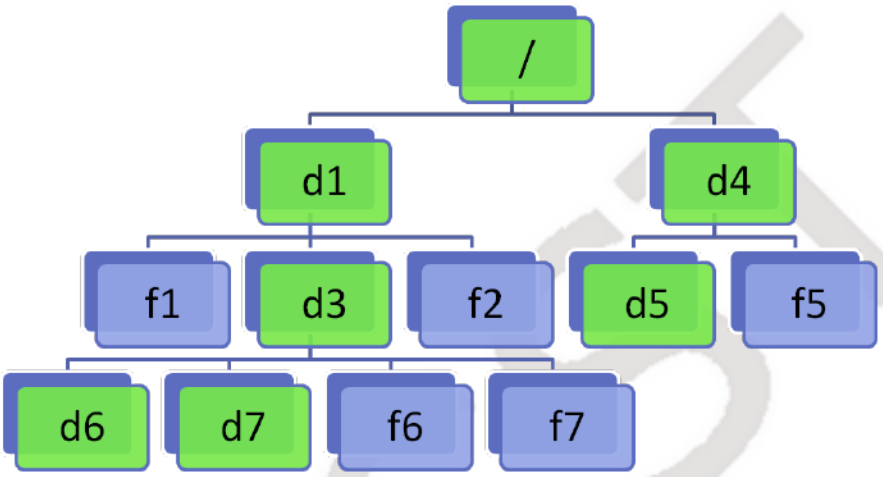


Figure a: The root file system

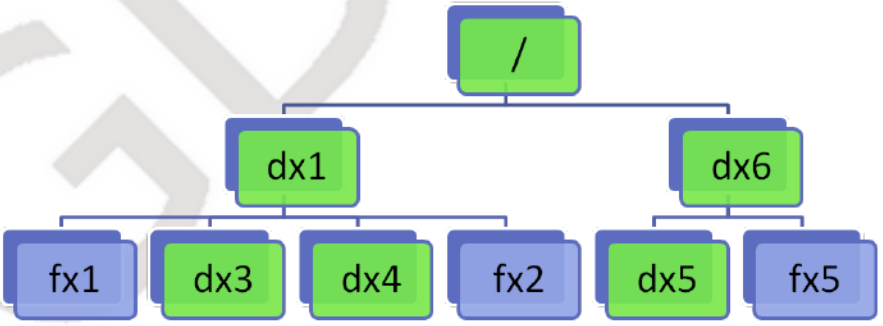


Figure b: File system on another device

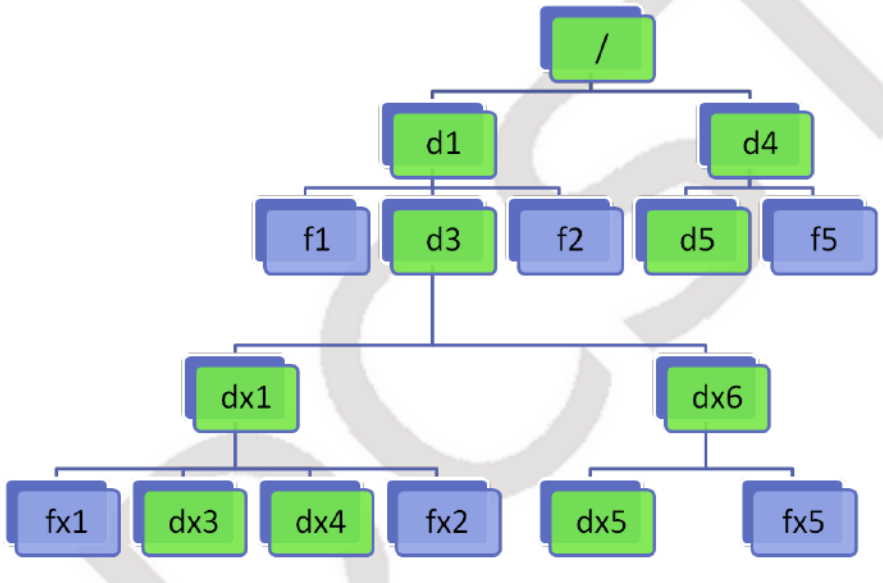


Figure c: After mounting the second file system on directory d3

Figure d: After unmounting the second file system from directory d3

Figure d: After unmounting the second file system from directory d3

However, the common practice is to mount file systems onto empty directories. In the default configuration, Ubuntu automatically detects other fixed devices in the system and shows them in the places menu and the left pane of the file browser. They are mounted when we first try to access them. The file browser shows a triangular icon alongside all mounted file systems other than the root file system. The file system can be unmounted by clicking on this icon. It may be mounted again when the user tries to access it again next time. The root file system cannot be unmounted. Removable devices are automatically mounted when they are inserted. Read-only media like optical disks can be unmounted by simply removing or ejecting them. Media on which writing is possible (like USB flash disks) must be unmounted by clicking the unmount icon in the file browser or by right clicking its icon on the desktop and selecting “safely remove device” option. This causes any data cached in main memory for improved performance to be flushed to the disk. A message at the end of this process announces that it is now safe to remove the device and the unmount icon disappears from besides the device’s entry in the file browser’s left pane. Only after this the device can be safely detached from the system. Failure to observe this procedure may result in loss of data or damage to the file system on the removable media. While this method of accessing the other storage devices in the system may sound unnecessarily complicated, it is more flexible and powerful and has several distinct advantages.

Not only local file systems, even the remote file systems can also be accessed in the same way. For example, Windows shares can be mounted on local directories using the SMB (Server Message Block) protocol and then be accessed just like local content. Similarly, NFS (Network File System), FTP server directories can also be mounted on a directory and accessed as a part of the file system.

Of course, all these procedures can also be carried out using commands. It is also possible to configure the system to mount some file systems at particular mount points automatically every time the system boots. By default, the system mounts other fixed devices in the system under the `/nt` directory and removable devices in the system in directories under the `/media` directory.

The Shell

- The shell acts as an interface between the user and the operating system
- The shell provides the user interface through which the users can start and stop programs
- The shell can be text-mode (Command Line Interface - CLI) or graphical mode (Graphical User Interface - GUI)
- File Explorer / Windows Explorer is the graphical shell that comes with Microsoft Windows
- cmd is the text-mode shell supplied along with Microsoft Windows
- Linux systems offer a variety of shells in both the GUI mode (GNOME Shell, KDE, Unity, XFCE, LXDE, MATE, etc.) and the text mode (sh, bash, ksh, csh, etc.)

The Linux Shells

Unix offers a variety of shells in both text mode and graphical mode. The graphical mode shells include the GNOME shell, Unity, KDE, XFCE, LXDE, MATE, etc. The commonly used text mode shells include sh (the original Bourne shell), ksh (Korn shell) csh (C Shell), bash (Bourne-Again SHell), etc.

The Bourne Shell

- **Internal commands** are built into the shell itself. There is no separate executable file for them
- **External commands** are not built into the shell. There is a separate executable file for them
- Search path
    - Since it is not practical to search the entire hard disk for the executable file corresponding to a possible external command entered by the user, only the directories in the search path are searched for the same. The search path is stored in the built-in shell variable PATH as string containing one or more directories separated by the : (colon) character
    - The current directory is not searched for a command if it is not in the search path
- CTRL-D at the beginning of a line indicates end-of-file (or end-of-input)
- CTRL-C is used to terminate the currently running foreground process
- CTRL-S is used to pause the terminal

- CTRL-Q is used to resume the terminal
- To copy text from the terminal, select the text using the mouse and press CTRL-SHIFT-C
- To paste text into the terminal, press CTRL-SHIFT-V. The text is inserted as if it had been typed by the user
- CTRL-L is used to redraw / refresh / clear the terminal
- The shell maintains the history of the commands entered earlier in memory. When the shell session terminates, this command history is saved in the file ~/.bash_history. The command history can be accessed using the UP / DOWN arrow keys at the prompt. It is also available in subsequent sessions and survives reboots
- The shell has its own full-fledged built-in programming language with variables, control structures, input, output, etc. It is a dynamic language in nature
- Case sensitivity
- Command completion using TAB and TAB-TAB

The Command Line

- **Components of the command line and word splitting** The command line contains one (or more) command(s), arguments to the command, operators, etc. The shell performs word splitting on the command line using white space (space / tab characters) as the separators
- In general, order of components of a command line is not important. However, there are several exceptions
- **Options**
  - **Short options** Short options consist of - (hyphen) followed by a single character
  - **Long options** Long options are specified using -- (two consecutive hyphens)
  - **Options with arguments** Some options have their own arguments. In such cases, the argument(s) must immediately follow the option
  - **Combining short options** Multiple short options can be combined by writing them immediately after a single hyphen: `ls -lh`. Only the last short option can have argument(s) in such case; and the argument, if any, is immediately specified after the combined options: `tar -czf backup.tar.gz directory`
- **Quoting and escaping** Strings on the command line can be specified

without using any quotes. However, if a string contains white space or some other character having a special meaning to the shell interpreter, the string must be enclosed in quotes or the white spaces / special characters must be escaped

- **Escaping** A character having special meaning to the shell can be escaped by preceding it with \ (backspace). This removes its special meaning. The special meaning of the backslash itself can also be removed by preceding it with another backslash `\\`
- **Single Quotes** Almost no processing is carried out inside strings enclosed in single quotes `'aaa bbb ccc'`
- **Double Quotes** Double quotes also behave like single quotes, but variable / parameter expansion *is* carried out inside double quotes: `$variable_name` is converted into the value of the variable. Hence if the value of variable *name* is *Scotty*, `echo 'Beam me up, $name'` outputs `Beam me up, $name` but `echo "Beam me up, $name"` outputs `Beam me up, Scotty`

File System Manipulation Commands

Note: *File system object* means either a file or a directory

- **pwd** (present working directory)
  - **pwd** Outputs the current working directory
- **cd** (change directory)
  - **cd directory** Changes the current directory to the given directory
  - **cd** changes the current directory to the home directory of the user. This behaviour is different from Windows where `cd` displays the current directory
- **ls** (list)
  - **ls** Outputs the names of the file system objects in the current directory
  - **ls arguments** Outputs the information about the argument(s). In case an argument is a file, its name is displayed. In case an argument is a directory, the names of file system objects inside the directory are displayed. If the argument file system object does not exist, an error message is output
  - **Options**
    - **-l** (*long* listing) Outputs a long listing with more information about the file system objects

        - **-d** (*directory*) When an argument is a directory, outputs information about the directory itself rather than its contents
- **mkdir** (make directory)
    - **mkdir directory** Creates the given directory if it does not exist
- **rmdir** (remove directory)
    - Removes (deletes) the given directory if it is empty
- **cat** (concatenate)
    - **cat file(s)** Outputs a concatenation of the given files. Also used to output a single file
- **cp** (copy)
    - **cp arguments** There must be at least two arguments. The last argument is the target. All earlier arguments are sources. *cp* does not copy directories by default. There are no warning or prompts if a file is going to be overwritten
        - **copy a file to another file** `cp source-file destination-file`
        - **copy one or more files to another directory** `cp file1 file2 file3 ... destination directory`
    - **Options**
        - **-r** (recursive) This option is used to copy a directory along with all its contents recursively
        - **-i** (interactive) Prompts before overwriting a file
- **rm** (remove)
    - **rm arguments** Removes (deletes) the given arguments. While the files and directories deleted from the GUI go into the trash, files and directories deleted from the command prompt are permanently lost. `rm` does not delete directories by default. It not prompt before deleting a file
    - **Options**
        - **-r** (recusrive) Directories in the arguments are deleted with all the contents recursively
        - **-i** (interactive) Prompts before deleting a file / directory
- **mv** (move)
    - **mv arguments** Moves or renames as per the number and types of arguments. Does not prompt before overwriting a file
        - **Rename a file or directory** `mv old-name new-name`
        - **Move a file or directory to another directory** `mv source target-directory`
    - **Options**

- **-i** (interactive) Prompts before overwriting a file

Shell Globbing Patterns

Pattern	Matches with
?	Any single character
*	Any number of any characters

Examples

Pattern	Matches with
p*	Anything starting with p
p*e	Anything that starts with p and ends with e, including pe
*~	All backup files
b??h	b, followed by exactly two characters, followed by h
b*h	Anything that starts with b and ends with h, including bh
dir*	Anything that starts with dir, including dir
dir?	dir followed by exactly one character

Plain Text Editors

- Plain text editors
  - Graphical editors
    - gedit / Text Editor (default)
    - gvim
  - Text mode editors
    - vi
    - vim (VI iMproved)
    - nano, pico
    - emacs (Editing MACroS)

The nano Editor

`nano` is a very simple text editor. It can be invoked by typing `nano filename` at

the prompt. Once the file is open, one can type any content one wishes. The following operations are supported:

- **CTRL+O ENTER** Save the current file
- **CTRL+X** Close the file and exit the nano editor. If the file is already saved, it will exit immediately. If the file is not saved, it will prompt `Save modified buffer?`. Press `y` for `yes`. Then it will prompt `File name to write:`. Press `ENTER` to accept the current file name. It will save the file and exit

The Shell Scripting Language of the *bash* Shell

Shell Variables

- The shell variables need not be declared. A variable is created automatically the first time it is assigned a value. All variables are of type *string*; but, in some contexts, may be interpreted as numbers. The variable assignment takes the form

```
variable_name=value
```

There must not be spaces around the = (assignment operator). Also, there is no `$` in front of the variable name when assigning value to it.

- The value of a variable can be accessed in the following ways:

```
$variable_name
${variable_name}
```

- It is ***not*** an error to attempt to access the value of an undefined variable. Such an attempt simply returns an empty string (a string of zero length)
- **Tip** Whenever you have a doubt that a variable may not exist, or that its value may be empty string or that its value may contain spaces in it, access its value using `"` (double quotes). For example, use `test -n "$v1"` instead of `test -n $v1`. You may also choose to always use double quotes
- **Built-in variables of the shell** The shell has a number of built-in variables. By convention, their names are in ALL_CAPS. For example, PATH, HOME,

SHELL, TERM, etc.

- **SHELL** : The current shell
- **HOME** : Home directory of the currently logged in user
- **PATH** : A : (clolon) separated list of directories that are searched for external commands
- **TERM** : The type of the current terminal
- **PS1** : The primary prompt
- **PS2** : The secondary prompt

The `test` Command and the `[` Builtin

- The test command
  - The command `test` tests some condition and returns exit status accordingly (0 means success / true and non-zero means failure / false). It does not produce any output
  - The spacial parameter `?` can be used to output the exit status of the last foreground process (`echo $?`)
  - The syntax `[ condition ]` can be used in place of test condition (`[` is a shell builtin)
  - There **must be** at least one space between each and every argument and `[` , `]`

```
echo $?
```

- String tests
  - **-z *string*** Returns true if *string* has zero length (empty string)
  - **-n *string*** Returns true if *string* has non-zero length
  - ***string1* = *string2*** Returns true if *string1* is equal to *string2*
  - ***string1* != *string2*** Returns true if *string1* is not equal to *string2*
- Integer tests
  - ***no1* -gt *no2*** Returns true if *no1* is greater than *no2*
  - ***no1* -ge *no2*** Returns true if *no1* is greater than or equal to *no2*
  - ***no1* -lt *no2*** Returns true if *no1* is less than *no2*
  - ***no1* -le *no2*** Returns true if *no1* is less than or equal to *no2*
  - ***no1* -eq *no2*** Returns true if *no1* is equal to *no2*
  - ***no1* -ne *no2*** Returns true if *no1* is not equal to *no2*

The Control Structures

- The *if* statement

```
if test condition1
then
    statement1
    statement2
    ...
elif test condition2
then
    statement3
    statement4
    ...
elif test condition3
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if [ condition1 ]
then
    statement1
    statement2
    ...
elif [ condition2 ]
then
    statement3
    statement4
    ...
elif [ condition3 ]
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

- The *while* statement

```
while test condition
do
    statement(s)
done
```

```
while [ condition ]
do
    statement(s)
done
```

- The *for* statement

```
for variable in white-space-separated-list
do
    statement(s)
done
```

- New line in syntax
    - A `;` (semicolon) can be used wherever the syntax requires a new line

- break and continue
  - `break` can be used in the loops to terminate the loop and jump to the next statement after the loop
  - `continue` can be used to terminate the current iteration of the loop and jump to the start of the loop
- The command `true` does nothing, but returns with an exit status of zero (success / true)
- The command `false` does nothing, but returns with a non-zero exit status (failure / false)

```
i=0
while true
do
    echo $i
    let i++
    if [ $i -ge 10 ]
    then
        break
    fi
done
```

Comments

- The **#** character is used for writing single-line comments
- All the characters starting from the # upto the end of the line are treated as comment and are ignored by the shell interpreter
- There is no syntax for multi-line comments

Output

- *echo*
  - `echo` is used to output its arguments
  - By default, echo appends a new line at the end of its output
  - To suppress that new line, use the `-n` option:

```
echo Your result is $result
echo "vaccination status: $vstatus"
echo -n "Enter your name: "
```

Input

- *read*
  - The `read` shell builtin can be used to read the values of one or more variables
  - `read x` reads a line from the input and assigns it to the variable x
  - The `-p` option can be used to display a prompt for reading

```
read x
read -p "Enter a number: " no
```

Integer Arithmetic

- *let*
  - `let` is a shell builtin for integer aritmetic
  - `let` expects an expression as a single argument (so the expression must not have spaces in it; or it should be quoted)
  - `let` supports the increment operator `++` and the decrement operator `--`. `let i++` adds `1` to the value of the variable `i`. It is same as writing `i=i+1`. `let i--` subtracts `1` from the value of the variable `i`. It is same as writing `i=i-1`.

```
let x=10*20
let x=(10+20)*20
let x=y++
let i++
```

Commands List

- **File System Handling**
  - **pwd** Displays the present working directory
  - **ls** Displays list of files
    - **-d** When an argument is a directory, display the directory (default is to display the contents of the directory)
    - **-h** Display human-readable size (20M, 1.5G, etc.)
    - **-l** Display a long list
  - **cat** Output standard input / a file / concatenate multiple files and output
  - **cd** Change directory
  - **mkdir** Make one or more directories

- **rmdir** Remove one or more directories (must be empty)
- **rm** Remove file(s) (and directories)
  - **-r** Remove file(s) and directories recursively
  - **-i** Remove interactively (prompt before deleting each file/directory)
- **cp** Copy file(s)
  - **-r** Copy file(s) and directories recursively
  - **-i** Copy interactively (prompt before deleting each file/directory)
- **mv** Move/rename files and directories
  - **-i** Move/rename files and directories interactively (prompt before deleting each file/directory)

- **Editing**
  - **nano**
- **Other Commands**
  - **clear** clear the screen
  - **echo** outputs its arguments followed by a newline
    - **-n** do not output a newline at the end
  - **read** Input a line and assign it to a variable
    - **-p** display the argument as a prompt if the input comes from a terminal
  - **test** test various conditions
    - **string tests**
      - **-z *str*** True of the string *str* is zero length (empty string)
      - **-n *str*** True of the string *str* is non-zero length
      - ***str1* = *str2*** True if *str1* is equal to *str2* (string comparison)
      - ***str1* != *str2*** True if *str1* is not equal to *str2* (string comparison)
    - **integer tests**
      - ***no1* -gt *no2*** True if *no1* is greater than *no2*
      - ***no1* -ge *no2*** True if *no1* is greater than or equal to *no2*
      - ***no1* -lt *no2*** True if *no1* is less than *no2*
      - ***no1* -le *no2*** True if *no1* is less than or equal to *no2*
      - ***no1* -eq *no2*** True if *no1* is equal to *no2* (integer comparison)
      - ***no1* -ne *no2*** True if *no1* is not equal to *no2* (integer comparison)
  - **man** show the manual page for the argument from the first section in which it exists