MINIPROJECT

TOPIC: FILE MANAAGEMENT USING SHELL PROGRAMMING

PROJECT DESCRIPTION:

Shell programming:

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text

A **shell script** is a computer program designed to be run by the Unix shell, a command-line interpreter.[1] The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text. A script which sets up the environment, runs the program, and does any necessary cleanup, logging, etc. is called a **wrapper**.

.

The typical Unix/Linux/POSIX-compliant installation includes the KornShell (`ksh`) in several possible versions such as ksh88, Korn Shell '93 and others. The oldest shell still in common use is the Bourne shell (`sh`); Unix systems invariably also include the C shell (`csh`), Bash(`bash`), a Remote Shell (`rsh`), a Secure Shell (`ssh`) for SSL telnet connections, and a shell which is a main component of the Tcl/Tkinstallation usually called `tclsh`; wish is a GUI-based Tcl/Tk shell.

In addition to the interactive mode, where the user types one command at a time, with immediate execution and feedback, Bash (like many other shells) also has the ability to run an entire script of commands, known as a "Bash shell script" (or "Bash script" or "shell script" or just "script"). A script might contain just a very simple list of commands — or even just a single command — or it might contain functions, loops, conditional constructs, and all the other hallmarks of imperative programming. In effect, a Bash shell script is a computer program written in the Bash programming language.

Shell scripting is the art of creating and maintaining such scripts.

Shell scripts can be called from the interactive command-line described above; or, they can be called from other parts of the system.

Shell scripts are commonly used for many system administration tasks, such as performing disk backups, evaluating system logs, and so on. They are also commonly used as installation scripts for complex programs. They are particularly suited to all of these because they *allow* complexity without *requiring* it: if a script just needs to run two external programs, then it can be a two-line script, and if it needs all the power and decision-making ability of a Turing-complete imperative programming language, then it can have that as well.

COMMANDS USED :

LS:

The Ls command is used to get a list of files and directories. Options can be used to get additional information about the files.

ls Syntax:
ls [options] [paths]

The ls command supports the following options:

- ls -a: list all files including hidden files. These are files that start with “.”..
- ls -l: list the files in long format i.e. with an index number, owner name, group name, size, and permissions.
- ls – o: list the files in long format but without the group name.
- ls -g: list the files in long format but without the owner name.
- ls -s: list the files along with their size.
- ls -t: sort the list by time of modification, with the newest at the top.
- ls -S: sort the list by size, with the largest at the top.
- ls -r: reverse the sorting order.

CD :

cd command in linux known as change directory command. It is used to change current working directory.

Syntax:

```
$ cd [directory]
```

The CD command supports the following options:

- `cd` by itself or `cd ~` will always put you in your home directory.
- `cd .` will leave you in the same directory you are currently in (i.e. your current directory won't change). This can be useful if your shell's internal code can't deal with the directory you are in being recreated; running `cd .` will place your shell in the recreated directory.
- `cd ~username` will put you in username's home directory.
- `cd dir` (without a `/`) will put you in a subdirectory; for example, if you are in `/usr`, typing `cd bin` will put you in `/usr/bin`, while `cd /bin` puts you in `/bin`.
- `cd ..` will move you up one directory. So, if you are `/usr/bin/tmp`, `cd ..` moves you to `/usr/bin`, while `cd ../..` moves you to `/usr` (i.e. up two levels). You can use this indirection to access subdirectories too. So, from `/usr/bin/tmp`, you can use `cd ../../local` to go to `/usr/local`.
- `cd -` will switch you to the previous directory. For example, if you are in `/usr/bin/tmp`, and go to `/etc`, you can type `cd -` to go back to `/usr/bin/tmp`. You can use this to toggle back and forth between two directorie

MKDIR:

SYNTAX:

```
mkdir [OPTION]... DIRECTORY...
```

DESCRIPTION

Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

The MKDIR command supports the following options:

TAG	DESCRIPTION
-m, --mode=MODE	set file mode (as in chmod), not a=rwx - umask
-p, --parents	no error if existing, make parent directories as needed
-v, --verbose	print a message for each created directory
-Z, --context=CTX	set the SELinux security context of each created directory to CTX
--help	display this help and exit
--version	output version information and exit
--kernelargs args	Adds args to the arguments appended on the kernel command line. If this is not specified mkbootdisk uses grubby to parse the arguments for the default kernel from grub.conf, if possible.
--size size	Uses size (in kilobytes) as the size of the image to use for the boot disk. If this is not specified, mkbootdisk will assume a standard 1.44Mb floppy device.

CAT:

SYNTAX:

```
cat [Options] [File]...
```

DESCRIPTION

Cat command concatenate FILE(s), or standard input, to standard output. With no FILE, or when FILE is -, it reads standard input.

The CAT command supports the following options:

TAG	Description
-A, --show-all	equivalent to -vET
-b, --number-nonblank	number nonblank output lines
-e	equivalent to -vE
-E, --show-ends	display $ at end of each line
-n, --number	number all output lines
-s, --squeeze-blank	never more than one single blank line
-t	equivalent to -vT
-T,--show-tabs	display TAB characters as ^I
-u	(ignored)
-v, --show-nonprinting	use ^ and M- notation, except for LFD and TAB. display this help and exit
--help	display this help and exit
--version	output version information and exit

ECHO:

echo is a built-in *command* in the *bash* and *C shells* that writes its *arguments* to *standard output*

The syntax for echo is

```
echo [option(s)] [string(s)]
```

The items in square brackets are optional. A *string* is any finite sequence of *characters* (i.e., letters, numerals, symbols and punctuation marks).

When used without any options or strings, echo returns a blank line on the display screen followed by the command prompt on the subsequent line. This is because pressing the ENTER key is a signal to the system to start a new line, and thus echo repeats this signal.

When one or more strings are provided as arguments, echo by default repeats those stings on the screen. Thus, for example, typing in the following and pressing the ENTER key would cause echo to repeat the phrase *This is a pen.* on the screen:

```
echo This is a pen.
```

RAED:

read is a builtin command of the Bash shell, which reads a line of text from standard input and splits it into words. These words can then be used as the input for other commands.

Syntax

```
read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N nchars]
```

```
[-p prompt] [-t timeout] [-u fd] [name ...] [name2 ...]
```

The READ command supports the following options:

```
-a array     assign the words read to sequential indices of the array
             variable ARRAY, starting at zero
      -d delim     continue until the first character of DELIM is read, rather
             than newline
      -e           use Readline to obtain the line in an interactive shell
      -i text      Use TEXT as the initial text for Readline
      -n nchars    return after reading NCHARS characters rather than waiting
             for a newline, but honor a delimiter if fewer than NCHARS
             characters are read before the delimiter
```