LABORATORY MANUAL FOR OPERATING SYSTEM

4^{TH} Semester

Diploma in Computer Science & Engineering

Department of Computer Science & Engineering

C. V. Raman Polytechnic

Bidya Nagar, Mahura, Janla, Bhubaneswar

(Affiliated to SCTE&VT: Approved by AICTE)

UNIX COMMANDS

AIM :

To study and execute the commands in Unix.

COMMAND:

1. Date Command:

This command is used to display the current data and time.

Syntax:

$date

$date +%ch

Options: -

a = Abbreviated weekday.

A = Full weekday.

b = Abbreviated month.

B = Full month.

c = Current day and time.

C = Display the century as a decimal number.

d = Day of the month.

D = Day in „mm/dd/yy‟ format

h = Abbreviated month day.

H = Display the hour.

L = Day of the year.

m = Month of the year.

M = Minute.

P = Display AM or PM

S = Seconds

T = HH:MM:SS format

u = Week of the year.

y = Display the year in 2 digit.

Y = Display the full year.

Z = Time zone .

To change the format :

Syntax :

$date „+%H-%M-%S‟

2. Calender Command:

This command is used to display the calendar of the year or the particular month of calendar year.

Syntax:

a.$cal <year>

b.$cal <month> <year>

Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

3. Echo Command:

This command is used to print the arguments on the screen .

Syntax:

$echo <text>

Multi line echo command:

To have the output in the same line, the following commands can be used.

Syntax:

$echo <text\>text

To have the output in different line, the following command can be used.

Syntax:

$echo “text

>line2

>line3”

4. Banner Command:

It is used to display the arguments in „#‟ symbol.

Syntax:

$banner <arguments>

5.’who’ Command:

It is used to display who are the users connected to our computer currently.

Syntax:

$who – option‟s

Options: -

H–Display the output with headers.

b–Display the last booting date or time or when the system was lastly rebooted.

6.’who am i’ Command:

Display the details of the current working directory.

Syntax:

$who am i

7.'tty' Command:

It will display the terminal name.

Syntax:

$tty

8.'Binary' Calculator Command:

It will change the „$" mode and in the new mode, arithematic operations such as +,-,*,/,%,n,sqrt(),length(),=, etc can be performed . This command is used to go to the binary calculus mode.

Syntax:

$bc operations

^d

$

1 base –input base

0 base – output base are used for base conversions.

Base:

Decimal = 1 Binary = 2 Octal = 8 Hexa = 16

9.'CLEAR' Command:

It is used to clear the screen.

Syntax:

$clear

10.'MAN' Command:

It helps us to know about the particular command and its options & working. It is like „help" command in windows.

Syntax:

$man <command name>

11. MANIPULATION Command:

It is used to manipulate the screen.

Syntax:

$tput <argument>

Arguments:

1. Clear – to clear the screen.
2. Longname – Display the complete name of the terminal.

3. SMSO – background become white and foreground become black color.

4. Rmso – background become black and foreground becomes white color.

5. Cop R C – Move to the cursor position to the specified location.

6. Cols – Display the number of columns in our terminals.

12. LIST Command:

It is used to list all the contents in the current working directory.

Syntax:

$ ls – options <arguments>

If the command does not contain any argument means it is working in the Current directory.

Options:

a– used to list all the files including the hidden files.

c– list all the files column wise.

d- list all the directories.

m- list the files separated by commas.

p- list files include „/" to all the directories.

r- list the files in reverse alphabetical order.

f- list the files based on the list modification date.

x-list in column wise sorted order.

DIRECTORY RELATED COMMANDS:

1. Present Working Directory Command:

To print the complete path of the current working directory.

Syntax:

```
 $pwd
```

2. MKDIR Command:

To create or make a new directory in a current directory .

Syntax:

```
$mkdir <directory name>
```

3. CD Command:

To change or move the directory to the mentioned directory .

Syntax:

```
$cd <directory name.
```

4. RMDIR Command:

To remove a directory in the current directory & not the current directory itself.

Syntax:

```
 $rmdir <directory name>
```

FILE RELATED COMMANDS:

1. CREATE A FILE:

To create a new file in the current directory we use CAT command.

Syntax:

```
$cat > <filename.
```

The > symbol is redirectory we use cat command.

2. DISPLAY A FILE:

To display the content of file mentioned we use CAT command without „>‟ operator.

Syntax:

```
$cat <filename.
```

Options

–s = to neglect the warning /error message.

3. COPYING CONTENTS:

To copy the content of one file with another.If file doesnot exist, a new file is created and if the file exists with some data then it is overwritten.

Syntax:

$ cat <filename source> >> <destination filename>

$ cat <source filename> >> <destination filename> it is avoid overwriting.

Options : -

-n content of file with numbers included with blank lines.

Syntax:

$cat –n <filename>

4. SORTING A FILE:

To sort the contents in alphabetical order in reverse order.

Syntax:

$sort <filename >

Option:

$ sort –r <filename>

5. COPYING CONTENTS FROM ONE FILE TO ANOTHER:

To copy the contents from source to destination file. So that both contents are same.

Syntax:

$cp <source filename> <destination filename>

$cp <source filename path > <destination filename path>

6. MOVE Command:

To completely move the contents from source file to destination file and to remove the source file.

Syntax:

$ mv <source filename> <destination filename>

7. REMOVE Command:

To permanently remove the file we use this command.

Syntax:

$rm <filename>

8. WORD Command:

To list the content count of no of lines, words, characters.

Syntax:

$wc<filename>

Options:

-c – to display no of characters.

-l – to display only the lines.

-w – to display the no of words.

9. LINE PRINTER:

To print the line through the printer, we use lp command.

Syntax:

$lp <filename>

10. PAGE Command:

This command is used to display the contents of the file page wise & next page can be viewed by pressing the enter key.

Syntax:

$pg <filename>

11. FILTERS AND PIPES

HEAD: It is used to display the top ten lines of file.

Syntax: $head<filename>

TAIL: This command is used to display the last ten lines of file.

Syntax: $tail<filename>

PAGE: This command shows the page by page a screen full of information is displayed after which the page command displays a prompt and passes for the user to strike the enter key to continue scrolling.

Syntax:

$ls –a\p

MORE: It also displays the file page by page .To continue scrolling with more command, press the space bar key.

Syntax:

$more<filename>

GREP: This command is used to search and print the specified patterns from the file.

Syntax:

$grep [option] pattern <filename>

SORT: This command is used to sort the datas in some order.

Syntax: $sort<filename>

PIPE: It is a mechanism by which the output of one command can be channeled into the input of another command.

Syntax:

$who | wc-l

Vi EDITOR COMMANDS

AIM:

To study the various commands operated in vi editor in UNIX.

DESCRIPTION:

The Vi editor is a visual editor used to create and edit text, files, documents and programs. It displays the content of files on the screen and allows a user to add, delete or change part of text . There are three modes available in the Vi editor , they are

1. Command mode
2. Input (or) insert mode.

Starting Vi:

The Vi editor is invoked by giving the following commands in UNIX prompt.

Syntax:

```
$vi <filename> (or)
$vi
```

This command would open a display screen with 25 lines and with tilt (~) symbol at the start of each line. The first syntax would save the file in the filename mentioned and for the next the filename must be mentioned at the end.

Options:

1. vi +n <filename> - this would point at the nth line (cursor pos).
2. vi –n <filename> - This command is to make the file to read only to change from one mode to another press escape key.

INSERTING AND REPLACING COMMANDS:

To move editor from command node to edit mode, you have to press the <ESC> key.

For inserting and replacing the following commands are used.

1. ESC a Command:

This command is used to move the edit mode and start to append after the current character.

Syntax:

```
 <ESC> a
```

2. ESC A COMMAND:

This command is also used to append the file , but this command append at the end of current line.

Syntax:

```
<ESC> A
```

3. ESC i Command:

This command is used to insert the text before the current cursor position.

Syntax:

<ESC> i

4. ESC I Command:

This command is used to insert at the beginning of the current line.

Syntax:

<ESC> I

5. ESC o Command:

This command is insert a blank line below the current line & allow insertion of contents.

Syntax:

<ESC> o

6. ESC O Command:

This command is used to insert a blank line above & allow insertion of contents.

Syntax:

<ESC> O

7. ESC r Command:

This command is to replace the particular character with the given characters.

Syntax:

<ESC> rx Where x is the new character.

8. ESC R Command:

This command is used to replace the particular text with a given text.

Syntax:

<ESC> R text

9. <ESC> s Command:

This command replaces a single character with a group of character .

Syntax:

<ESC> s

10. <ESC> S Command:

This command is used to replace a current line with group of characters.

Syntax:

<ESC> S

CURSOR MOVEMENT IN Vi:

1. <ESC> h:

This command is used to move to the previous character typed. It is used to move to left of the text . It can also used to move character by character (or) a number of characters.

Syntax:

<ESC> h – to move one character to left.

<ESC> nh – to move „n" character to left.

2. <ESC> l:

This command is used to move to the right of the cursor (ie) to the next character. It can also be used to move the cursor for a number of characters.

Syntax:

<ESC> l – single character to right.

<ESC> nl - „n" characters to right.

3. <ESC> j:

This command is used to move down a single line or a number of lines.

Syntax:

<ESC> j – single down movement.

<ESC> nj – „n" times down movement.

4. <ESC> k:

This command is used to move up a single line or a number of lines.

Syntax:

<ESC> k – single line above.

<ESC> nk – „n" lines above.

5. ENTER (OR) N ENTERS:

This command will move the cursor to the starting of next lines or a group of lines mentioned.

Syntax:

<ESC> enter <ESC> n enter.

6. <ESC> + Command:

This command is used to move to the beginning of the next line.

Syntax:

<ESC> + <ESC> n+

7.<ESC> - Command :

This command is used to move to the beginning of the previous line.

Syntax:

<ESC> - <ESC> n8.<ESC> 0 :

This command will bring the cursor to the beginning of the same current line.

Syntax:

<ESC> 0

9. <ESC> $:

This command will bring the cursor to the end of the current line.

Syntax:

<ESC> $

10. <ESC> ^:

This command is used to move to first character of first lines.

Syntax:

<ESC> ^

11. <ESC> b Command:

This command is used to move back to the previous word (or) a number of words.

Syntax:

<ESC> b <ESC>nb

12. <ESC> e Command:

This command is used to move towards and replace the cursor at last character of the word (or) no of words.

Syntax:

<ESC> e <ESC>ne

13. <ESC> w Command:

This command is used to move forward by a single word or a group of words.

Syntax:

<ESC> w <ESC> nw

DELETING THE TEXT FROM Vi:

1. <ESC> x Command:

To delete a character to right of current cursor positions, this command is used.

Syntax:

<ESC> x <ESC> nx

2. <ESC> X Command:

To delete a character to left of current cursor positions, this command is used.

Syntax:

<ESC> X <ESC> nX

3. <ESC> dw Command:

This command is to delete a single word or number of words to right of current cursor position.

Syntax:

<ESC> dw <ESC> ndw

4. db Command:

This command is to delete a single word to the left of the current cursor position.

Syntax:

<ESC> db <ESC> ndb

5. <ESC> dd Command:

This command is used to delete the current line (or) a number of lines below the current line.

Syntax:

<ESC> dd <ESC> ndd

6. <ESC> d$ Command:

This command is used to delete the text from current cursor position to last character of current line.

Syntax:

<ESC> d$

SAVING AND QUITING FROM Vi:-

1. <ESC> w Command:

To save the given text present in the file.

Syntax:

<ESC> : w

2. <ESC> q! Command:

To quit the given text without saving.

Syntax:

<ESC> :q!

3. <ESC> wq Command:

This command quits the vi editor after saving the text in the mentioned file.

Syntax:

<ESC> :wq

4. <ESC> x Command:

This command is same as „wq‟ command it saves and quit.

Syntax:

<ESC> :x

5. <ESC> q Command:

This command would quit the window but it would ask for again to save the file.

Syntax:

<ESC> : q

UNIX SHELL PROGRAMMING COMMANDS.

AIM:

To study about the Unix Shell Programming Commands.

INTRODUCTION:

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled. Common Shells. C-Shell - csh : The default on teaching systems Good for interactive systems Inferior programmable features Bourne Shell - bsh or sh - also restricted shell - bsh : Sophisticated pattern matching and file name substitution Korn Shell : Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode Thomas C-Shell - tcsh : Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shell.

01. SHELL KEYWORDS:

echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

02. General things SHELL

The shbang line The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

EXAMPLE

```
#!/bin/sh
```

Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

EXAMPLE

```
# this text is not
# interpreted by the shell
```

Wildcards There are some characters that are evaluated by the shell in a special way. They are called shell met characters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

EXAMPLE

Filename expansion:

```
rm *; ls ??; cat file[1-3];
```

Quotes protect met character:

```
echo "How are you?"
```

03. SHELL VARIABLES:

Shell variables change during the execution of the program .The C Shell offers a command "Set" to assign a value to a variable.
For example:

```
% set myname= Fred
% set myname = "Fred Bloggs"
% set age=20
```

A $ sign operator is used to recall the variable values.
For example:
% echo $myname will display Fred Bloggs on the screen A @ sign can be used to assign the integer constant values.
For example:

```
%@myage=20
%@age1=10
%@age2=20
%@age=$age1+$age2
%echo $age
```

List variables

```
% set programming_languages= (C LISP)
% echo $programming _languages
C LISP
% set files=*.*
% set colors=(red blue green)
% echo $colors[2]
blue
% set colors=($colors yellow)/add to list
```

Local variables Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.
EXAMPLE

```
variable name=value
name="John Doe"
x=5
```

Global variables Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.
EXAMPLE

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Extracting values from variables To extract the value from variables, a dollar sign is used.
EXAMPLE

```
echo $variable name
echo $name
echo $PATH
```

Rules: -

1. A variable name is any combination of alphabets, digits and an underscore („-„);
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variables names should be of any reasonable length.
5. Variables name are case sensitive. That is , Name, NAME, name, Name, are all different variables.

04. EXPRESSION Command:

To perform all arithmetic operations.

Syntax:

Var = „expr$value1" + $ value2"

Arithmetic The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations.

EXAMPLE

```
n=`expr 5 + 5`
echo $n
```

Operators The Bourne shell uses the built-in test command operators to test numbers and strings.

EXAMPLE

Equality:
= string
!= string
-eq number
-ne number
Logical:
-a and
-o or
! not
Logical:
AND &&
OR ||
Relational:
-gt greater than
-ge greater than, equal to
-lt less than
-le less than, equal to
Arithmetic :
+, -, *, /, %

Arguments (positional parameters) Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.

EXAMPLE

At the command line:

```
$ scriptname arg1 arg2 arg3 ...
```

In a script:

echo $1 $2 $3 Positional parameters
echo $* All the positional paramters
echo $# The number of positional parameters

05. READ Statement:
To get the input from the user.
Syntax:
read x y
(no need of commas between variables)
06. ECHO Statement:
Similar to the output statement. To print output to the screen, the echo command is used. Wildcards must be escaped with either a backslash or matching quotes.
Syntax:
Echo “String” (or) echo $ b(for variable).
EXAMPLE
echo "What is your name?"
Reading user input The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept muliple variable names. Each variable will be assigned a word.
EXAMPLE
echo "What is your name?"
read name
read name1 name2 ...
6. CONDITIONAL STATEMENTS:
The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.
Syntax:
1. if
This is used to check a condition and if it satisfies the condition if then does the next action , if not it goes to the else part.
2. If…else
Syntax:
If cp $ source $ target
Then
Echo File copied successfully
Else
Echo Failed to copy the file.
3. nested if
here sequence of condition are checked and the corresponding performed accordingly.
Syntax:
if condition
then

```
command
 if condition
 then
 command
 else
command
 fi
fi
```

4. case ….. esac

This construct helps in execution of the shell script based on Choice.

EXAMPLE

The if construct is:

```
if command
then
 block of statements
fi
-------------------------------------
if [ expression ]
then
 block of statements
fi
-------------------------------------
```

The if/else/else if construct is:

```
if command
then
 block of statements
elif command
then
 block of statements
elif command
then
```

The case command construct is:

```
case variable_name in
 pattern1)
 statements
 ;;
 pattern2)
 statements
 ;;
 pattern3)
 ;;
 *) default value
 ;;
esac
```

```
case "$color" in
 blue)
 echo $color is blue
 ;;
 green)
 echo $color is green
 ;;
block of statements
else
 block of statements
fi
------------------------------
if [ expression ]
then
 block of statements
elif [ expression ]
then
 block of statements
elif [ expression ]
then
 block of statements
else
 block of statements
fi
--------------------------------------
 red|orange)
 echo $color is red or orange
 ;;
 *) echo "Not a color" # default
esac
```

The if/else construct is:

```
if [ expression ]
then
 block of statements
else
 block of statements
fi
--------------------------------------
```

07. LOOPS

There are three types of loops: while, until and for. The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed. The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false. The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

EXAMPLE

```
while command
do
block of statements
done
------------
while [ expression ]
do
block of statements
done
until command for variable in word1 word2 word3 ...
do do
block of statements block of statements
done done
------------
until [ expression ]
do
 block of statements
done
------------
until control command
do
commands
done
```

08. Break Statement:

This command is used to jump out of the loop instantly, without waiting to get the control command.

09. ARRAYS (positional parameters)

The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in set command, and the words are accessed by position. Up to nine positions are allowed.The built-in shift command shifts off the first word on the left-hand side of the list. The individual words are accessed by position values starting at 1.

EXAMPLE

```
set word1 word2 word3
echo $1 $2 $3 Displays word1, word2, and word3
set apples peaches plums
shift Shifts off apples
echo $1 Displays first element of the list
```

echo $2 Displays second element of the list
echo $* Displays all elements of the list
Command substitution To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, backquotes are used.

EXAMPLE

```
variable_name=`command`
echo $variable_name
now=`date`
echo $now
echo "Today is `date`"
```

10. FILE TESTING

The Bourne shell uses the test command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth.

EXAMPLE

-d File is a directory
-f File exists and is not a directory
–r Current user can read the file
–s File is of nonzero size
–w Current user can write to the file
–x Current user can execute the file

```
#!/bin/sh
1 if [ –f file ]
 then
 echo file exists
 fi
2 if [ –d file ]
 then
 echo file is a directory
 fi
3 if [ -s file ]
 then
 echo file is not of zero length
 fi
4 if [ -r file -a -w file ]
 then
 echo file is readable and writable
 fi
```

11. EXECUTION OF SHELL SCRIPT:

1. By using change mode command
2. $ chmod u + x sum.sh
3. $ sum.sh
 or
 $ sh sum.sh

Experiment 1:

Write a Shell script to print the command line arguments in reverse order.

Program:

```
#!/bin/sh
if [ $# -eq 00 ]
then
   echo "no arguments given"
   exit
fi
echo "Total number of arguments: $#"
echo "The arguments are: $*"
echo "The arguments in reverse order:"
rev=" "
for i in $*
do
  rev=$i" "$rev
done
echo $rev
```

Output 1

```
$ ./print-arguments-in-reverse.sh
no arguments given
```

Output 2

```
$ ./print-arguments-in-reverse.sh 9 abc hello
Total number of arguments: 3
The arguments are: 9 abc hello
The arguments in reverse order:
hello abc 9
```

Experiment 2:

Write a Shell script to check whether the given number is palindrome or not.

Program:

```
echo enter n
read n
num=0
on=$n
while [ $n -gt 0 ]
do
num=$(expr $num \* 10)
k=$(expr $n % 10)
num=$(expr $num + $k)
n=$(expr $n / 10)
done
if [ $num -eq $on ]
then
echo palindrome
else
echo not palindrome
fi
```

Output:

```
$ enter n
$ 121
$ palindrome
```

Experiment 3:

Write a Shell script to sort the given array elements in ascending order using bubble sort.

Program:

```
#!/bin/bash
echo "enter maximum number"
read n
echo "enter Numbers in array:"
for (( i = 0; i < $n; i++ ))
do
   read nos[$i]
done
echo "Numbers in an array are:"
for (( i = 0; i < $n; i++ ))
do
   echo ${nos[$i]}
done
for (( i = 0; i < $n ; i++ ))
do
   for (( j = $i; j < $n; j++ ))
   do
      if [ ${nos[$i]} -gt ${nos[$j]}  ]; then
      t=${nos[$i]}
      nos[$i]=${nos[$j]}
      nos[$j]=$t
      fi
   done
done
echo -e "\nSorted Numbers "
for (( i=0; i < $n; i++ ))
do
   echo ${nos[$i]}
done
```

Output:

```
$ bash bubblesort.sh
Enter maximum number
6
Enter Numbers in array:
2
4
7
8
22
3
Numbers in an array are:
2
4
7
8
22
3
Sorted Numbers
2
3
4
7
8
22
```

Experiment 4:

Write a Shell script to perform sequential search on a given array elements.

Program:

```
echo Enter the number of elements:
read n
echo Enter the array elements:
for (( i=1; i<=n; i++ ))
do
   read a[$i]
done
echo Enter the element to be searched:
read item
j=1
while [ $j -lt $n -a $item -ne ${a[$j]} ]
do
   j=`expr $j + 1`
done
if [ $item -eq ${a[$j]} ]
then
   echo $item is present at location $j
else
   echo "$item is not present in the array."
fi
```

Output:

```
Enter the number of elements:
5
Enter the array elements:
2
10
5
3
11
Enter the element to be searched:
5
5 is present at location 3
```

Experiment 5:

Write a Shell script to perform binary search on a given array elements.

Program:

```
echo “Enter the limit:”
read n
echo “Enter the numbers”
for(( i=0 ;i<n; i++ ))
do
read m
a[i]=$m
done
for(( i=1; i<n; i++ ))
do
for(( j=0; j<n-i; j++))
do
if [ ${a[$j]} -gt ${a[$j+1]} ]
then
t=${a[$j]}
a[$j]=${a[$j+1]}
a[$j+1]=$t
fi
done
done
echo “Sorted array is”
for(( i=0; i<n; i++ ))
do
echo “${a[$i]}”
done
echo “Enter the element to be searched :”
read s
l=0
c=0
u=$(($n-1))
while [ $l -le $u ]
```

```
do
mid=$(((( $l+$u ))/2 ))
if [ $s -eq ${a[$mid]} ]
then
c=1
break
elif [ $s -lt ${a[$mid]} ]
then
u=$(($mid-1))
else
l=$(($mid+1))
fi
done
if [ $c -eq 1 ]
then
echo “Element found at position $(($mid+1))”
else
echo “Element not found”
fi
```

Output:

```
Enter the limit: 5
Enter the numbers
4 2 6 3 7
Sorted array is
2 3 4 6 7
 Enter the element to be searched: 7
Element found at position 5
```

Experiment 6:

Write a Shell script to accept any two file names and check their file permissions.

Program:

```
#!/bin/bash
# Check if two arguments are provided
if [ $# -ne 2 ]; then
   echo "Usage: $0 <file1> <file2>"
   exit 1
fi
file1=$1
file2=$2
# Check if files exist
if [ ! -e "$file1" ] || [ ! -e "$file2" ]; then
   echo "One or both files do not exist."
   exit 1
fi
# Check file permissions for the first file
echo "File permissions for $file1:"
if [ -r "$file1" ]; then
   echo "Read permission is granted."
else
   echo "Read permission is not granted."
fi
if [ -w "$file1" ]; then
   echo "Write permission is granted."
else
   echo "Write permission is not granted."
fi
if [ -x "$file1" ]; then
   echo "Execute permission is granted."
else
   echo "Execute permission is not granted."
fi
```

```
echo "--------------------------------"
# Check file permissions for the second file
echo "File permissions for $file2:"
if [ -r "$file2" ]; then
   echo "Read permission is granted."
else
   echo "Read permission is not granted."
fi
if [ -w "$file2" ]; then
   echo "Write permission is granted."
else
   echo "Write permission is not granted."
fi
if [ -x "$file2" ]; then
   echo "Execute permission is granted."
else
   echo "Execute permission is not granted."
fi
```

Output:

```
$ ./check_permissions.sh file1.txt file2.txt
File permissions for file1.txt:
Read permission is granted.
Write permission is granted.
Execute permission is not granted.
--------------------------------
File permissions for file2.txt:
Read permission is granted.
Write permission is not granted.
Execute permission is not granted.
```

Experiment 7:

Write a Shell script to read a path name, create each element in that path e.g: a/b/c i.e., ‘a’ is directory in the current working directory, under ‘a’ create ‘b’, under ‘b’ create ‘c’.

Program:

```
#!/bin/bash
echo -n "Enter the path name: "
read path
if [ -z "$path" ]; then
   echo "Path is empty. Exiting."
   exit 1
fi
IFS='/' read -ra elements <<< "$path"
current_dir="."
for element in "${elements[@]}"; do
   current_dir="$current_dir/$element"
   if [ -d "$current_dir" ]; then
      echo "Directory '$current_dir' already exists."
   else
      mkdir "$current_dir"
      echo "Created directory: $current_dir"
   fi
done
echo "Path creation complete."
```

Output:

```
$ ./create_path.sh
Enter the path name: a/b/c
Created directory: ./a
Created directory: ./a/b
Created directory: ./a/b/c
Path creation complete.
```

Experiment 8:

Write a Shell script to illustrate the case-statement.

Program:

```
#!/bin/bash
# Prompt the user to enter a fruit name
echo -n "Enter a fruit name: "
read fruit
# Using case statement to check the fruit name
case $fruit in
   "apple")
      echo "You entered an apple. It's a red or green fruit."
      ;;
   "banana")
      echo "You entered a banana. It's a yellow fruit."
      ;;
   "orange" | "mandarin" | "clementine")
      echo "You entered an orange family fruit. It's typically orange in color."
      ;;
   "grape" | "kiwi")
      echo "You entered a grape or kiwi. These are small fruits."
      ;;
   *)
      echo "Unrecognized fruit."
      ;;
esac
```

Output:

```
$ ./fruit_script.sh
Enter a fruit name: apple
You entered an apple. It's a red or green fruit.
```

Experiment 9:

Write a Shell script to accept the file name as arguments and create another shell script, which recreates these files with its original contents.

Program:

```
#!/bin/bash
# Check if a file name is provided as an argument
if [ $# -eq 0 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi
filename=$1
# Check if the file exists
if [ ! -e "$filename" ]; then
    echo "File '$filename' does not exist."
    exit 1
fi
# Generate the shell script to recreate the file
output_script="recreate_$filename.sh"
echo "#!/bin/bash" > "$output_script"
echo "" >> "$output_script"
echo "# Recreate file '$filename' with its original contents" >> "$output_script"
echo "cat <<'EOF' > \"$filename\"" >> "$output_script"
# Append the original contents of the file to the script
cat "$filename" >> "$output_script"
echo "EOF" >> "$output_script"
# Make the generated script executable
chmod +x "$output_script"
echo "Shell script to recreate '$filename' has been generated: $output_script"
```

Output:

```
cat <<'EOF' > "example.txt"
Contents of the original file go here.
Replace this with the actual content of 'example.txt'.
EOF
```

Experiment 10:

Write a Shell script to demonstrate Terminal locking.

Program:

```
#!/bin/bash
stty -echo
while true
do
    clear
    echo "Enter the password:"
    read -s pass1  # -s flag for silent input, hiding characters
    echo "Re-enter the password:"
    read -s pass2
    if [[ "$pass1" == "$pass2" ]]; then
        echo "Terminal locked"
        echo "To unlock, enter the password"
        typed_pass=""
        while [ "$typed_pass" != "$pass2" ]
        do
            read -s typed_pass
        done
        echo "Terminal unlocked"
        stty echo  # Re-enable terminal echo
        exit
    else
        echo "Password mismatch, please retype it"
        sleep 2  # Add a delay to make it more user-friendly
    fi
done
```

Output:

```
Enter the password:
********
Re-enter the password:
********
Terminal locked
To unlock, enter the password
********
Terminal unlocked
```

Experiment 11:

Write a Shell script to accept the valid login name, if the login name is valid then print its home directory else an appropriate message.

Program:

```
#!/bin/bash
# Prompt the user to enter a login name
echo -n "Enter a login name: "
read login_name
# Check if the login name is valid
if [ -z "$login_name" ]; then
    echo "Error: Login name cannot be empty."
    exit 1
fi
# Get the home directory of the provided login name
home_directory=$(getent passwd "$login_name" | cut -d: -f6)
# Check if the home directory exists
if [ -z "$home_directory" ]; then
    echo "Error: Invalid login name or home directory not found for '$login_name'."
else
    echo "Home directory for '$login_name': $home_directory"
fi
```

Output:

```
./check_login.sh
Enter a login name: non_existent_user
Error: Invalid login name or home directory not found for 'non_existent_user'.
```

Experiment 12:

Write a Shell script to read a file name and change the existing file permissions.

Program:

```
#!/bin/bash
echo -n "Enter a file name: "
read filename
if [ -z "$filename" ]; then
    echo "Error: File name cannot be empty."
    exit 1
fi
if [ ! -e "$filename" ]; then
    echo "Error: File '$filename' does not exist."
    exit 1
fi
echo "Current file permissions for '$filename':"
ls -l "$filename"
echo "-------------------------------------"
echo -n "Enter new file permissions (e.g., 644): "
read new_permissions
chmod "$new_permissions" "$filename"
echo "Updated file permissions for '$filename':"
ls -l "$filename"
```

Output:

```
Enter a file name: your_file.txt
Current file permissions for 'your_file.txt':
-rw-r--r-- 1 user user 1024 Jan 10 15:30 your_file.txt
-------------------------------------
Enter new file permissions (e.g., 644): 755
Updated file permissions for 'your_file.txt':
-rwxr-xr-x 1 user user 1024 Jan 10 15:30 your_file.txt
```

Experiment 13:

Write a Shell script to print current month calendar and to replace the current day number by ‘*’ or ‘**’ respectively.

Program:

```
#!/bin/bash
# Get the current day number
current_day=$(date +"%d")
cal
echo "Replace the current day number with '*' or '**'"
echo -n "Enter '1' for '*' or '2' for '**': "
read choice
case $choice in
    1)      cal | sed -e "s/\b$current_day\b/*/"      ;;
    2)      cal | sed -e "s/\b$current_day\b/**/"      ;;    *)
        echo "Invalid choice. Exiting."
        exit 1      ;;
esac
```

Output:

```
    January 2024
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
Replace the current day number with '*' or '**'
Enter '1' for '*' or '2' for '**': 2
    January 2024
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
```

Experiment 14:

Write a Shell Script to display a menu consisting of options to display disk space, the current users logged in, total memory usage, etc. (using functions.)

Program:

```
#!/bin/bash
# Function to display disk space
display_disk_space() {
   df -h
}
# Function to display currently logged-in users
display_logged_in_users() {
   who
}
# Function to display total memory usage
display_memory_usage() {
   free -h
}
# Main menu function
main_menu() {
   echo "==== Menu ===="
   echo "1. Display Disk Space"
   echo "2. Display Logged-in Users"
   echo "3. Display Memory Usage"
   echo "4. Exit"
   echo -n "Enter your choice (1-4): "
   read choice
   case $choice in
      1)   display_disk_space   ;;
      2)   display_logged_in_users   ;;
      3)   display_memory_usage    ;;
      4)   echo "Exiting."
         exit 0    ;;     *)
         echo "Invalid choice. Please enter a number between 1 and 4."            ;;
   esac
```

```
}
# Main loop to keep displaying the menu
while true
do
   main menu
done
```

Output:

```
==== Menu ====
1. Display Disk Space
2. Display Logged-in Users
3. Display Memory Usage
4. Exit
Enter your choice (1-4): 1
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        30G   10G   18G  36% /

==== Menu ====
1. Display Disk Space
2. Display Logged-in Users
3. Display Memory Usage
4. Exit
Enter your choice (1-4): 2
username   pts/0        2022-01-01 08:00 (192.168.1.1)
anotheruser pts/1        2022-01-01 09:30 (192.168.1.2)

==== Menu ====
1. Display Disk Space
2. Display Logged-in Users
3. Display Memory Usage
4. Exit
Enter your choice (1-4): 3
              total        used        free      shared  buff/cache   available
Mem:           7.7G        2.1G        3.4G        512M        2.2G        5.2G
Swap:          2.0G        0B          2.0G

==== Menu ====
1. Display Disk Space
2. Display Logged-in Users
3. Display Memory Usage
4. Exit
Enter your choice (1-4): 4
Exiting.
```

Experiment 15:

Write a C-program to fork a child process and execute the given Linux commands.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        printf("Child process executing...\n");
        execlp("ls", "ls", "-l", NULL);
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        printf("Parent process waiting for the child...\n");
        wait(NULL);
        printf("Parent process exiting.\n");
    }
    return 0;
}
```

Output:

```
Parent process waiting for the child...
Child process executing...
total 12
-rwxr-xr-x 1 user user 8672 Jan 10 15:30 fork_exec
Parent process exiting.
```

Experiment 16:

Write a C-program to fork a child process, print owner process ID and its parent process ID.

Program:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
    } else if (pid == 0) {
        printf("Child process:\n");
        printf("PID: %d\n", getpid());
        printf("Parent PID: %d\n", getppid());
    } else {
        printf("Parent process:\n");
        printf("PID: %d\n", getpid());
        printf("Child PID: %d\n", pid);
    }
    return 0;
}
```

Output:

```
Parent process:
PID: 1234
Child PID: 1235

Child process:
PID: 1235
Parent PID: 1234
```

Experiment 17:

Write a C-program to prompt the user for the name of the environment variable, check its validity and print an appropriate message.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char var_name[50];
    printf("Enter the name of the environment variable: ");
    scanf("%s", var_name);
    char *var_value = getenv(var_name);
    if (var_value != NULL) {
        printf("The value of %s is: %s\n", var_name, var_value);
    } else {
        printf("Environment variable %s does not exist.\n", var_name);
    }
    return 0;
}
```

Output:

```
Enter the name of the environment variable: HOME
The value of HOME is: /home/your_username
```

Experiment 18:

Write a C-program to READ details of N students such as student name, reg number, semester and age. Find the eldest of them and display his details.

Program:

```
#include <stdio.h>
struct Student {
    char name[50];
    int regNumber;
    int semester;
    int age;
};
int main() {
    int n;
    printf("Enter the number of students: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid number of students. Exiting.\n");
        return 1;
    }
    struct Student students[n];
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for student %d:\n", i + 1);
        printf("Name: ");
        scanf("%s", students[i].name);
        printf("Registration Number: ");
        scanf("%d", &students[i].regNumber);
        printf("Semester: ");
        scanf("%d", &students[i].semester);
        printf("Age: ");
        scanf("%d", &students[i].age);
    }
    int eldestIndex = 0;
    for (int i = 1; i < n; i++) {
        if (students[i].age > students[eldestIndex].age) {
            eldestIndex = i;
        }
    }
    printf("\nDetails of the eldest student:\n");
    printf("Name: %s\n", students[eldestIndex].name);
    printf("Registration Number: %d\n", students[eldestIndex].regNumber);
    printf("Semester: %d\n", students[eldestIndex].semester);
    printf("Age: %d\n", students[eldestIndex].age);
    return 0;
}
```

Output:

```
Enter the number of students: 3

Enter details for student 1:
Name: Alice
Registration Number: 101
Semester: 2
Age: 20

Enter details for student 2:
Name: Bob
Registration Number: 102
Semester: 3
Age: 22

Enter details for student 3:
Name: Charlie
Registration Number: 103
Semester: 1
Age: 19

Details of the eldest student:
Name: Bob
Registration Number: 102
Semester: 3
Age: 22
```