UNIX Shell Programming (15CS35)

Solution
Internal Assessment- II
November - 2016

1. **Explain significance of following commands (2x5=10 M)**

i. cp ?????? progs

Wild card ? matches any single character, hence the above command (cp) copies files whose names are six in length to progs directory

ii. ls *.[xyz]*

Wild card * matches any number of characters, hence the above command (ls) lists all the files having extension as either x, or y or z.

iii. ls jones[0-9][0- 9][0-9]

In the above command the character class[0-9] matches any digit between 0 to 9. Hence the above command lists all the files beginning with jones and having last three characters as any digit between 0 to 9.

iv. echo *

The above command lists all the file in the current directory.

v. cp foo foo*

The above command copies the file foo to file called foo*. Here the wild card * loses its meaning.

1. **Write UNIX commands for the following (2 X 5= 10 M)**

i. Find and replace all the occurrences of **unix** with **UNIX** in the file after confirming the user.

```
: 1,$s/unix/UNIX/gc
```

ii. List all the files in the current directory which names are having exactly 5 characters and any number of characters in their extension.

```
ls ?????.*
```

iii. Copy all files stored in /home/vtu with .c, .cpp and .java extensions to progs sub-directory in current directory

```
cp /home/vtu/*.{c,cpp,java} ./progs
```

iv. Delete all files containing **agra** or **agar** in their file names.

rm *ag[ra][ar]*

v. Display contents of current directory and its sub-directories.

ls –R */*

2) Explain the three sources of standard input and standard output. Explain the two special files in UNIX (6+4=10 M)

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device: Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:

- The keyboard, the default source.
- A file using redirection with the < symbol.
- Another program using a pipeline.

The standard output can represent three possible destinations:

- The terminal, the default destination.
- A file using the redirection symbols > and >>.
- As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input

1 – Standard output

2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.

Examples: Assuming file2 doesn‟t exist, the following command redirects the standard output to file myOutput and the standard error to file myError.

```
 $ ls –l file1 file2 1>myOutput 2>myError
```

To redirect both standard output and standard error to a single file use:

```
$ ls –l file1 file2 1>| myOutput 2>| myError OR

$ ls –l file1 file2 1> myOutput 2>& 1
```

/dev/null and /dev/tty : Two special files

/dev/null: If you would like to execute a command but don't like to see its contents on the screen, you may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

2) Define a shell script. Explain the following statements with syntax and examples
i) if ii) case iii) for (1+9=10 M)

Shell Script: When a group of commands have to be executed regularly , they should be stored in a file, and the file itself executed as a shell script or a shell program

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

Syntax

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
else
  Statement(s) to be executed if expression is not true
fi
```

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
  echo "a is equal to b"
else
  echo "a is not equal to b"
fi
```

<u>Case:</u>

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
esac
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

<u>for: Looping with a List</u>

for is also a repetitive structure.
Synatx:

```
for variable in list

do

            Commands

done
```

list here comprises a series of character strings. Each string is assigned to variable specified.
Example:

```
for file in ch1 ch2; do
> cp $file ${file}.bak
```

```
    >        echo  $file  copied  to
    $file.bak done
```
Output:

```
ch1  copied  to  ch1.bak  ch2
copied to ch2.bak
```

Sources of list:

- **List from variables**: Series of variables are evaluated by the shell before executing the loop
  Example:
  ```
  $ for var in $PATH $HOME; do echo "$var" ; done
  ```
  Output:
  ```
  /bin:/usr/bin;/home/local/bin;
  /home/user1
  ```
- **List from command substitution**: Command substitution is used for creating a list. This is used when list is large.
  Example:
  ```
  $ for var in `cat clist`
  ```
- **List from wildcards**: Here the shell interprets the wildcards as filenames. Example:
  ```
  for  file  in  *.htm  *.html  ;  do  sed
          's/strong/STRONG/g
          s/img src/IMG SRC/g' $file > $$ mv $$
          $file
  done
  ```
- **List from positional parameters**:
  Example: emp.sh
  ```
  #! /bin/sh
  for pattern in "$@"; do
  grep "$pattern" emp.lst || echo "Pattern $pattern not found" done
  ```

3) Explain the three modes of vi editor with a neat diagram and list the commands in each mode (10 M)

Input Mode – Entering and Replacing Text It is possible to display the mode in which is user is in by typing,

:set showmode Messages like INSERT MODE, REPLACE MODE, CHANGE MODE, etc will appear in the last line. Pressing „i" changes the mode from command to input mode. To append text to the right of the cursor position, we use a, text. I and A behave same as i and a, but at line extremes I inserts text at the beginning of line. A appends text at end of line. o opens a new line below the current line

• r replacing a single character

• s replacing text with s

• R replacing text with R

• Press esc key to switch to command mode after you have keyed in text Some of the input mode commands are: COMMAND i a I A o O r s S

Saving Text and Quitting – The ex Mode When you edit a file using vi, the original file is not distributed as such, but only a copy of it that is placed in a buffer. From time to time, you should save your work by writing the buffer contents to disk to keep the disk file current. When we talk of saving a file, we actually mean saving this buffer. You may also need to quit vi after or without saving the buffer. Some of the save and exit commands of the ex mode is:

Command	Action
:W	saves file and remains in editing mode
:x	saves and quits editing mode
:wq	saves and quits editing mode
:w	save as
:w!	save as, but overwrites existing file
:q	quits editing mode
:q!	quits editing mode by rejecting changes made
: sh	escapes to UNIX shell
:recover	recovers file from a crash

3) a. What are hard links? Explain two applications of hard links? What are the two main disadvantages of the hard link? (1+2+2=5M)

The link count is displayed in the second column of the listing. This count is normally 1, but the following files have two links,

```
-rwxr-xr-- 2 kumar metal 163 Jull 13 21:36 backup.sh -rwxr-xr-- 2
kumar metal 163 Jul 13 21:36 restore.sh
```

All attributes seem to be identical, but the files could still be copies. It's the link count that seems to suggest that the files are linked to each other. But this can only be confirmed by using the –i option to ls.

```
ls -li backup.sh restore.sh
```

```
478274  -rwxr-xr--  2  kumar  metal163  jul  13  21:36  backup.sh
478274 -rwxr-xr-- 2 kumar metal163 jul 13 21:36 restore.sh
```

ln: Creating Hard Links

A file is linked with the ln command which takes two filenames as arguments (cp command). The command can create both a hard link and a soft link and has syntax similar to the one used by cp. The following command links emp.lst with employee:

```
ln emp.lst employee
```

The –i option to ls shows that they have the same inode number, meaning that they are actually one end the same file:

```
ls -li emp.lst employee
```

```
29518 -rwxr-xr-x 2 kumar metal 915 may 4 09:58 emp.lst 29518
-rwxr-xr-x 2 kumar metal 915 may 4 09:58 employee
```

The link count, which is normally one for unlinked files, is shown to be two. You can increase the number of links by adding the third file name emp.dat as:

```
ln employee emp.dat ; ls -l emp*
```

```
29518  -rwxr-xr-x  3  kumar  metal  915  may  4  09:58  emp.dat  29518
-rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.lst 29518 -rwxr-xr-x 3
kumar metal 915 may 4 09:58 employee
```

You can link multiple files, but then the destination filename must be a directory. A file is considered to be completely removed from the file system when its link count drops to zero. ln returns an error when the destination file exists. Use the –f option to force the removal of the existing link before creation of the new one

Where to use Hard Links

```
ln data/ foo.txt input_files
```

It creates link in directory *input_files*. With this link available, your existing programs will continue to find foo.txt in the *input_files* directory. It is more convenient to do this that modifies all programs to point to the new path. Links provide some protection

against accidental deletion, especially when they exist in different directories. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program and to a shell

script. A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called.

We can't have two linked filenames in two file systems and we can't link a directory even within the same file system. This can be solved by using symbolic links (soft links).

b) Explain these commands with example i) umask ii) cut (5+5=10M)

UMASK (User Mask or User file creation MASK) is the default permission or base permissions given when a new is created.
The default value of umask is 022
In UNIX, the default file creation value is 666. 6 is 4+2(read + write). Permission 666 means 6 for the User, 6 for the group and 6 for others. Hence, a new file creation by default is meant to have read and write permission for User, group and others. This is the place where the umask comes into the picture. It is a kind of filter wherein we can choose to retain or block some of the default permissions from being applied on the file.
Assume we create a file say "file1". The permissions given for this file will be the result coming from the substraction of the umask from the default value :

```
 Default:666
 umask:022
---------------
Result:644
```

644 is the permission to be given on the file "file1". 644 means read and write for the User(<u>6</u>44), read only for the group(6<u>4</u>4) and others(64<u>4</u>).

The same rule is applied while creating a directory as well
cut – slitting a file vertically

It is used for slitting the file vertically. head -n 5 emp.lst | tee shortlist will select the first five lines of emp.lst and saves it to *shortlist.* We can cut by using -c option with a list of column numbers, delimited by a comma (cutting columns).

```
cut -c 6-22,24-32 shortlist
```

```
cut -c -3,6-22,28-34,55- shortlist
```

The expression 55- indicates column number 55 to end of line. Similarly, -3 is the same as 1-3.

Most files don’t contain fixed length lines, so we have to cut fields rather than columns

(cutting fields).

-d for the field delimiter -f for
the field list

```
cut -d \ | -f 2,3 shortlist | tee cutlist1
```

will display the second and third columns of *shortlist* and saves the output in *cutlist1.* here | is escaped to prevent it as pipeline character

- To print the remaining fields, we have

```
cut –d \ | -f 1,4- shortlist > cutlist2
```

4)a.Write commands for the followings:

i) Select the lines longer than 100 and smaller than 150 characters using grep

```
grep -E '^.{101,149}$' Filename
```

ii) Select the lines from the file that have only the string UNIX

```
grep ^UNIX$ filename
```

b) How do these expressions differ? (6 M)

i)[0-9]* and [0-9[0- 9]* : [0-9]* matches zero or more occurrences of any digit whereas [0-9[0- 9]* matches one or more occurrences

ii) ^[^^] and ^^^ : ^[^^] matches everything and ^^^ matches nothing
iii) AB\{4\} and (AB)\{4\} : AB\{4\}matches a A and at least 4 occurrences of B whereas (AB)\{4\} matches 4 occurrences of AB

4) a. What is the exit status of a command and where is it stored

To terminate a program exit is used. Nonzero value indicates an error condition.

Example 1:

```
$ cat foo
Cat: can’t open foo
```

Returns nonzero exit status. The shell variable $? Stores this status.

Example 2:

```
grep director emp.lst > /dev/null:echo $? 0
```

Exit status is used to device program logic that braches into different paths depending on success or failure of a command

4) b. Explain the special parameters used by the shell

Variable	**Description**
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

5) Write a shell program to create option based execution on users choice. Options include i) list of users ii) list of processes. iii) list of files iv) current date v) print current directory vi) clear screen

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

```
# menu.sh : Uses case to offer vi-item menu

echo "              MENU\n
      1.  Users of system\n 2. Processes of users\n 3. List of files\n 4. Today's date\n
      5.  Current directory\n 6. Clear the screen\n 7. Quit \n Enter your option\c:"
read choice

case "$choice" in

      1)  who ;;
      2)  ps –f ;;
      3)  ls –l ;;
      4)  date ;;
      5)  pwd ;;
      6)  clear ;;
      7)  exit ;;
      *)  echo "Invalid option\n" ;;

esac
```

5) a. Write a shell script to add n numbers using a while loop (5 X 2= 10 M)

```
#!/bin/sh
# sum.sh : To calculate sum of n numbers
echo -n "Enter number : "
read n

# store single digit
i=1
# store number of digit
sum=0
# use while loop to caclulate the sum of all digits
while [ $i -le $n ]
  do
   sum=$(( $sum + $i )) # calculate sum of digit
        i=$(( $i + 1 ))
  done
echo  "Sum of all digit  is $sum"
```

b. Write a shell script to illustrate the usage of set and shift commands

```
#!/bin/sh
# emp.sh : Script using shift
```

```
case $# in

0|1) echo “Insufficient arguments\n”; exit 2 ;;
  *) flname=$1
        shift
        for pattern in “$@” ; do
                grep “$pattern” $flname || echo “Pattern $pattern not found”
        done ;;
esac
```