Q.2 a. Explain the structure of UNIX file system. (8)

Answer:

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A UNIX file system is a collection of files and directories that has the following properties:

- It has a root directory (/) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an inode.
- By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0 and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
- It is self contained. There are no dependencies between one file system and any other.

b. What is the difference between internal and external commands? (4+4)

Answer:

Since `ls` is a program or file having an independent existence in the `/bin` directory (or `/usr/bin`), it is branded as an **external command**. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by `PATH`. Take for instance the **echo** command:

```
$ type echo
echo is a shell builtin
```

echo isn't an external command in the sense that, when you type echo, the shell won't look in its `PATH` to locate it (even if it is there in `/bin`). Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which **echo** is a member, are known as **internal commands.**

You must have noted that it's the shell that actually does all this work. This program starts running for you when you log in, and dies when you log out. The shell is an external command with a difference; it possesses its own set of internal commands. So if a command exists both as an internal command of the shell as well as an external one (in `/bin` or `/usr/bin`), the shell will accord top priority to its own internal command of the same name.

This is exactly the case with **echo**, which is also found in `/bin`, but rarely ever executed because the shell makes sure that the internal **echo** command takes precedence over the external. We'll take up the shell in detail later.

Q.3 a. Which command is used to change permission associated to File/Directories? List and explain methods to change permission of File/Directories? List and explain methods to change permission of File/Directories. (8)

Answer:

To change file or directory permissions, you use the chmod (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

$ls -l testfile

-rwxrwxr-- 1
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx 1 amrood users 1024 Nov 2 00:10 testfile

Using chmod with Absolute Permissions:

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Then each example chmod command from the preceding table is run on testfile, followed by ls -l

$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x 1 amrood users 1024 Nov 2 00:10 testfile

Changing Owners and Groups:

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files:

1. chown: The chown command stands for "change owner" and is used to change the owner of a file.
2. chgrp: The chgrp command stands for "change group" and is used to change the group of a file.

Changing Group Ownership:

The chrgp command changes the group ownership of a file. The basic syntax is as follows:

$ chgrp group filelist

The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

b. How will you accomplish the following in vi? (8)
(i) Combine two consecutive lines into one.
(ii) Display line numbers.
(iii) Display the file in read only mode.
(iv) Save the file and quit vi.

Answer:

i) When you want to merge two lines into one, position the cursor anywhere on the first line, and press `J` to join the two lines.

ii) `:set number`

`iii).` file in read-only mode, enter either:

`$ vi -R file,  $ view file`

iv) :wq

Q.4 a. Write a Shell script to find the total and average of four numbers to be read from the user with respect to Unix. (8)

Answer:

```
echo Enter four integers with space between
read a b c d
sum=`expr $a + $b + $c + $d`
avg=`expr $sum / 4`
echo Sum=$sum
echo Average=$avg.$dec
```

b. Define the following terms: (8)
(i) file
(ii) process
(iii) Multi-user
(iv) Multitasking

Answer:

i) Unix does not impose or provide any internal file structure. This implies that from the point of view of the operating system, there is only one file type. The structure and interpretation thereof is entirely dependent on how the file is interpreted by software. Unix does however have some special files. These special files can be identified by the ls -l command which displays the type of the file in the first alphabetic letter of the file system permissions field. A normal (regular) file is indicated by a hyphen-minus '-'.

ii) When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system. Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the ls command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program. The operating system tracks processes through a five digit ID number known as the pid or process ID . Each process in the system has a unique pid.

iii) multi-user more than one user can use the machine at a time supported via terminals (serial or network connection)

iv) multi-tasking more than one program can be run at a time.

Q.5 a. Explain the three timestamps associated with a file. (3+3+3)

Answer:

A UNIX file has three time stamps associated with it. In this section, we'll be discussing just two of them (the first two):

- Time of last file modification — *Shown by* `ls -l`
- Time of last access — *Shown by* `ls -lu`
- Time of last inode modification — *Shown by* `ls -lc`

Whenever you write to a file, the time of last modification is updated in the file's inode. A directory can be modified by changing its entries—by creating, removing and renaming files in the directory. Note that changing a file's contents only changes its last modification time but not that of the directory. **`ls -l`** displays the last modification time.

A file also has an access time, i.e., the last time someone read, wrote or executed the file. This time is distinctly different from the modification time that gets set only when the contents of the file are changed. For a directory, the access time is changed by a read operation only; creating or removing a file or doing a "cd" to a directory doesn't change its access time. The access time is displayed when **`ls -l`** is combined with the **`-u`** option.

Even though **`ls -l`** and **`ls -lu`** show the time of last modification and access, respectively, the sort order remains standard, i.e. ASCII. However, when you add the `-t` option to `-l` or `-lu`, the files are actually displayed in *order* of the respective time stamps:

`ls -lt`	Displays listing in order of their modification time
`ls -lut`	Displays listing in order of their access time

Knowledge of a file's modification and access times is extremely important for the system administrator. Many of the tools used by him look at these time stamps to decide whether a particular file will participate in a backup or not. A file is often incorrectly stamped when extracting it (using an option) from a backup with a file restoration utility (like **`tar`** or **`cpio`**). If that has happened to you, you can use **`touch`** to reset the times to certain convenient values without actually modifying or accessing the file. **`touch`** is discussed next.

11.6.1 touch: Changing the Time Stamps

As has just been discussed, you may sometimes need to set the modification and access times to predefined values. The **`touch`** command changes these times, and is used in the following manner:

`touch` *options expression filename(s)*

When **`touch`** is used without *options* or *expression*, both times are set to the current time. The file is created if it doesn't exist:

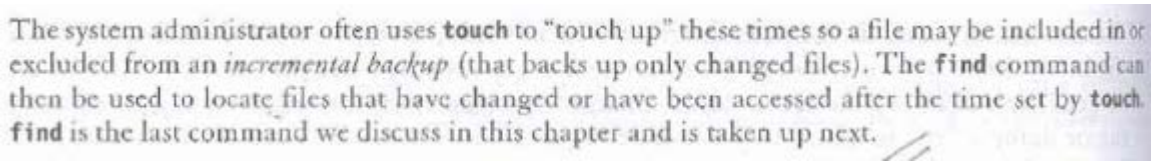
```
touch emp.lst                                   Creates file if it doesn't exist
```

When **`touch`** is used without options but with *expression*, it changes both times. The *expression* consists of an eight-digit number using the format *MMDDhhmm* (month, day, hour and minute). Optionally, you can suffix a two- or four-digit year string:

```
$ touch 03161430 emp.lst ; ls -l emp.lst
-rw-r--r--   1 kumar    metal         870 Mar 16 14:30 emp.lst
$ ls -lu emp.lst
-rw-r--r--   1 kumar    metal         870 Mar 16 14:30 emp.lst
```

It's also possible to change the two times individually. The `-m` and `-a` options change the modification and access times, respectively:

```
$ touch -m 02281030 emp.lst ; ls -l emp.lst
-rw-r--r--   1 kumar    metal         870 Feb 28 10:30 emp.lst
$ touch -a 01261650 emp.lst ; ls -lu emp.lst
-rw-r--r--   1 kumar    metal         870 Jan 26 16:50 emp.lst
```


The system administrator often uses **touch** to "touch up" these times so a file may be included in or excluded from an *incremental backup* (that backs up only changed files). The **find** command can then be used to locate files that have changed or have been accessed after the time set by **touch**. **find** is the last command we discuss in this chapter and is taken up next.

b. Describe the content of /etc/passwd. (7)

Answer:

/etc/passwd file is used to keep track of every registered user that has access to a system.

The /etc/passwd file is a colon-separated file that contains the following information:

- User name
- Encrypted password
- User ID number (UID)
- User's group ID number (GID)
- Full name of the user (GECOS)
- User home directory
- Login shell

Q.6 a. What do these commands do? (8)

(i) grep a b c
(ii) grep <HTML> foo
(iii) grep*foo
(iv) grep "letter" abc

Answer:

i. searches for any lines in files b and c that contains the character "a"
ii. error occurs because it thinks you're redirecting "<" and ">"; correct: grep "\<HTML\>" foo
iii. searches for any lines in the file foo that contains zero or more of the character "*"
iv. searches for any lines in the file xyz that contains "letter" word

b. Explain with examples use of + and ? in grep command. (4+4)

Answer:

The ERE set includes two special characters, + and ?. They are often used in place of the * to *restrict* the matching scope. They signify the following:

+ — Matches one or more occurrences of the previous character.

? — Matches zero or one occurrence of the previous character.

In both cases, the emphasis is on the *previous* character. This means that b+ matches b, bb, bbb, etc., but unlike b*, it doesn't match nothing. The expression b? matches either a single instance of b or nothing. These characters restrict the scope of match as compared to the *.

Using this extended set, you can now have a different regular expression for matching Agarwal and aggarwal. Note that the character g occurs only once or twice. So, gg? now restricts the expansion to one or two gs only. This time we need to use **grep**'s -E option to use an ERE:

```
$ grep -E "[aA]gg?arwal" emp.lst
2476|anil aggarwal    |manager  |sales      |01/05/59|5000
3564|sudhir Agarwal   |executive|personnel  |06/07/47|7500
```

The + is a pretty useful character too. When you are looking for a multiword string like #include <stdio.h>, but don't know how many spaces separate the #include and <stdio.h>, you can use the expression #include +<stdio.h> to match them all. This expression matches the following patterns:

```
#include <stdio.h>      #include  <stdio.h>      #include      <stdio.h>
```

And if you are not sure whether there's a space between # and include, include the ? in the expression

```
# ?include +<stdio.h>
```

But there could be tabs here instead of spaces as well, so how does one handle them?

Q.7 a. Explain use of expr command. (8)

Answer:

The Bourne shell can check whether an integer is greater than another, but it doesn't have any computing features at all. It has to rely on the external **expr** command for that purpose. This command combines two functions in one:

- Performs arithmetic operations on integers.
- Manipulates strings.

We'll use **expr** to perform both these functions, but with not-very-readable code when it comes to string handling. If you are using the Korn shell or Bash, you have better ways of handling these things *(21.7)*, but you must also understand the helplessness of Bourne. It's quite possible that you have to debug someone else's script which contains **expr**.

14.9.1 Computation

expr can perform the four basic arithmetic operations as well as the modulus (remainder) function:

```
$ x=3 y=5                                   Multiple assignments without a ;
$ expr 3 + 5
8
$ expr $x - $y
-2
$ expr 3 \* 5                               Asterisk has to be escaped
15
$ expr $y / $x
1                                           Decimal portion truncated
$ expr 13 % 5
3
```

The operand, be it +, -, * etc., must be enclosed on either side by whitespace. Observe that the multiplication operand (*) has to be escaped to prevent the shell from interpreting it as the filename metacharacter. Since **expr** can handle only integers, division yields only the integral part.

expr is often used with command substitution to assign a variable. For example, you can set a variable z to the sum of two numbers:

```
$ x=6 y=2 ; z=`expr $x + $y`
$ echo $z
8
```

Perhaps the most common use of **expr** is in incrementing the value of a variable. All programming languages have a shorthand method of doing that, and it is natural that UNIX should also have its own:

```
$ x=5
$ x=`expr $x + 1`                           This is the same as C's x++
$ echo $x
6
```

If you are using the Korn shell or Bash, then you can turn to Section 21.5 for a discussion on the **let** statement that both shells use to handle computation.

14.9.2 String Handling

Though **expr**'s string handling facilities aren't exactly elegant, Bourne shell users hardly have any choice. For manipulating strings, **expr** uses two expressions separated by a colon. The string to be worked upon is placed on the left of the :, and a regular expression is placed on its right. Depending on the composition of the expression, **expr** can perform three important string functions:

- Determine the length of the string.
- Extract a substring.
- Locate the position of a character in a string.

The Length of a String The length of a string is a relatively simple matter; the regular expression .* signifies to **expr** that it has to print the number of characters matching the pattern, i.e., the length of the entire string:

```
$ expr "abcdefghijkl" : '.*'                    Space on either side of : required
12
```

Here, **expr** has counted the number of occurrences of any character (.*). This feature is useful in validating data entry. Consider that you want to validate the name of a person accepted through the keyboard so that it doesn't exceed, say, 20 characters in length. The following **expr** sequence can be quite useful for this task:

```
while echo "Enter your name: \c" ; do                    echo always returns true
    read name
    if [ `expr "$name" : '.*'` -gt 20 ] ; then
        echo "Name too long"
    else
        break                                            break terminates a loop
    fi
done
```

Extracting a Substring **expr** can extract a string enclosed by the escaped characters \(and \). If you wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
$ stg=2003
$ expr "$stg" : '..\(..\)'                       Extracts last two characters
03
```

Note the pattern group \(..\). This is the tagged regular expression (TRE) used by **sed** *(13.11.3)*, but it is used here with a somewhat different meaning. It signifies that the first two characters in the value of $stg have to be ignored and two characters have to be extracted from the third character position. (There's no \1 and \2 here.)

Locating Position of a Character **expr** can also return the location of the first occurrence of a character inside a string. To locate the position of the character d in the string value of $stg, you have to count the number of characters which are not d ([^d]*), followed by a d:

```
$ stg=abcdefgh ; expr "$stg" : '[^d]*d'
4
```

expr duplicates some of the features of the **test** statement, and also uses the relational operators in the same way. They are not pursued here because **test** is a built-in feature of the shell, and is consequently faster. The Korn shell and Bash have built-in string handling facilities; they don't need **expr**.

b. What is the exit status of a command? What is its normal value and in which parameter is its value stored? (8)

Answer:

The exit status is an integer that represents the success or failure of a command. If has the value 0 when the command executes successfully and is stored in the parameter $?

Q.8 a. What is the use of following built in variables in awk? (8)

(i) NR

(ii) FS
(iii) NF
(iv) FILENAME

Answer:

awk has several built-in variables (Table 18.2). They are all assigned automatically, though it is also possible for a user to reassign some of them. You have already used NR, which signifies the record number of the current line. We'll now have a brief look at some of the other variables.

The FS *Variable* As stated elsewhere, **awk** uses a contiguous string of spaces as the default field delimiter. FS redefines this field separator, which in the sample database happens to be the |. When used at all, it must occur in the BEGIN section so that the body of the program knows its value before it starts processing:

```
BEGIN { FS="|" }
```

This is an alternative to the -F option which does the same thing.

The OFS *Variable* When you used the **print** statement with comma-separated arguments, each argument was separated from the other by a space. This is **awk**'s default output field separator, and can be reassigned using the variable OFS in the BEGIN section:

```
BEGIN { OFS="~" }
```

When you reassign this variable with a ~ (tilde), **awk** will use this character for delimiting the **print** arguments. This is a useful variable for creating lines with delimited fields.

The NF *Variable* NF comes in quite handy for cleaning up a database of lines that don't contain the right number of fields. By using it on a file, say empx.lst, you can locate those lines not having six fields, and which have crept in due to faulty data entry:

```
$ awk  'BEGIN { FS = "|" }
> NF != 6 {
> print "Record No ", NR, "has ", NF, " fields"}' empx.lst
```

```
Record No 6 has 4 fields
Record No 17 has 5 fields
```

The FILENAME *variable* FILENAME stores the name of the current file being processed. Like grep and **sed**, **awk** can also handle multiple filenames in the command line. By default, **awk** doesn't print the filename, but you can instruct it to do so:

```
'$1 < 4000 { print FILENAME, $0 }'
```

With FILENAME, you can devise logic that does different things depending on the file that is processed.

Table 18.2 Built-in Variables Used by **awk**

Variable	*Function*
NR	Cumulative number of lines read
FS	Input field separator
OFS	Output field separator
NF	Number of fields in current line
FILENAME	Current input file
ARGC	Number of arguments in command line
ARGV	List of arguments

b. Explain in detail the simple awk filtering. (8)

Answer:

a filter action is the most efficient way to limit its scope. In AWK scripts, the action specified by such a conditional filter occurs only if the specified pattern matches the record in question.

The format for a conditional filter rule is as follows:

pattern { action }

The action here is a series of statements just like any other filter rule. The pattern can be blank (in which case it matches every record), or it can contain any combination of regular expressions or relational expressions. These two types of expressions are briefly explained in the following sections.

Q.9 a. How do you double-space a file with perl? (8)

Answer:

Double space a file.

perl -pe '$\="\n"'

This one-liner double spaces a file. There are three things to explain in this one-liner. The "-p" and "-e" command line options, and the "$\" variable.

```
while (<>) {
   # your program goes here
} continue {
   print or die "-p failed: $!\n";
}
```

This construct loops over all the input, executes your code and prints the value of "$_". This way you can effectively modify all or some lines of input. The "$_" variable can be explained as an anonymous variable that gets filled with the good stuff.

The "$\" variable is similar to ORS in Awk. It gets appended after every "print" operation. Without any arguments "print" prints the contents of "$_" (the good stuff).

In this one-liner the code specified by "-e" is '$\="\n"', thus the whole program looks like this:

```
while (<>) {
   $\ = "\n";
} continue {
   print or die "-p failed: $!\n";
}
```

There is actually no need to set "$\" to newline on each line. It was just the shortest possible one-liner that double-spaced the file. Here are several others that do the same:

perl -pe 'BEGIN { $\="\n" }'

This one sets the "$\" to newline just once before Perl does anything (BEGIN block gets executed before everything else).

perl -pe '$_ .= "\n"'

This one-liner is equivalent to:

```
while (<>) {
   $_ = $_ . "\n"
} continue {
   print or die "-p failed: $!\n";
}
```

It appends another new-line at the end of each line, then prints it out.

The cleanest and coolest way to do it is probably use the substitution "s///" operator:

perl -pe 's/$/\n/'

It replaces the regular expression "$" that matches at the end of line with a newline, effectively adding a newline at the end.

b. Explain the split and join functions available in perl? (4+4)

Answer:

19.10 split: SPLITTING INTO A LIST OR ARRAY

CGI programmers using **perl** need to understand two important array handling functions—**split** and **join**. **split** breaks up a line or expression into fields. These fields are assigned either to variables or an array. Here are the two syntaxes:

```
($var1, $var2, $var3.....    ) = split(/sep/,stg) ;
@arr = split(/sep/,stg) ;
```

split takes up to three arguments but is usually used with two. It splits the string *stg* on *sep*, but here *sep* can be a literal character or a regular expression (which could expand to multiple characters). *stg* is optional, and in its absence, $_ is used as default. The fields resulting from the split are assigned either to the variables *$var1*, *$var2* and so on, or to the array *@arr*.

19.10.1 Splitting into Variables

We'll now use the first syntactical form in our next program, **3_numbers.pl** (Fig. 19.7), to assign three numbers, taken from the keyboard, to a set of variables.

To understand how **split** breaks up the variable $numstring and assigns three new variables, let's run this program twice:

```
$ 3_numbers.pl
Enter three numbers: [Enter]                    Nothing entered
Nothing entered
```

```
#!/usr/bin/perl
# Script: 3_numbers.pl - Splits a string on whitespace
#
print("Enter three numbers: " ) ;
chop($numstring = <STDIN>) ;
die("Nothing entered\n") if ($numstring eq "") ;
($f_number, $s_number, $l_number) = split(/ /, $numstring) ;
print("The last, second and first numbers are ") ;.
print("$l_number, $s_number and $f_number.\n" ) ;
```

Fig. 19.7 **3_numbers.pl**

```
$ 3_numbers.pl
Enter three numbers: 123 345 567
The last, second and first numbers are 567, 345 and 123.
```

When the three numbers are entered, $numstring acquires the value 123 345 567\n, from where the newline is subsequently chopped off. **split** acts on this string using a *single* space as delimiter, and breaks it up into three variables.

19.10.2 Splitting into an Array

What do you do when a line contains a large number of fields? In that case, it's better to split it into an array rather than variables. The following statement fills up the array @thislist:

```
@thislist = split(/:/, $string) ;
```

$string is often the last line read, in which case we can replace it with $_, or rather, drop it altogether:

```
@thislist = split(/:/) ;                    split uses $_ by default
```

In the next program, **rep1.pl** (Fig. 19.8), we print some specific fields from the sample database with the last name shown before the first name. We need to use **split** twice—first on the | delimiter and then again on the space that separates the first and last names. The program selects the first four lines and prints a total of the salary field for the selected lines.

The second **split** was used to enable the reversal of the first and last names, with a comma between them. We also use the range operator (..) to print only the first four lines, and $. to print the current line number as serial number:

```
$ rep1.pl emp.lst
  1 shukla, a.k.          sales       6000
  2 sharma, jai           production  7000
```

```
#!/usr/bin/perl
# Script: rep1.pl - Uses split twice; prints with first and last name reversed
#
while (<>) {
   chop;
   @field = split (/\|/) ;   # $_ is used by default
   if (1..4) {               # Lines 1 to 4
       $dept = $field[3] ; $name = $field[1] ; $salary = $field[5] ;
       ($f_name, $l_name) = split(/ +/, $name);
       $name = $l_name . ", " . $f_name ;      # Reusing $name
       $totsal += $salary ;
       printf( "%3d %-20s %-11s %4d\n", $., $name, $dept, $salary) ;
   }
}
printf("%35s %5d\n", "Total Salary: ", $totsal) ;
```

Fig. 19.8 **rep1.pl**

```
  3 chakrobarty, sumit    marketing    6000
  4 sengupta, barun       personnel    7800
                    Total Salary:    26800
```

split can also be used without an explicit assignment, in which case it populates the built-in array, @_:

```
split (/:/) ;                                    Fills up the array @_
```

split gets shortened further. The array @_ has the elements $_[0], $_[1] and so forth. You should get used to this form also as you'll see it used in many programs.

Note: When the return value of **split** is not *explicitly* assigned to variables or an array, the built-in array, @_, is automatically assigned. Also, when **split** is used with the null string (//) as delimiter, @_ stores *each* character of the string as a separate element.

19.11 join: JOINING A LIST

The **join** function acts in an opposite manner to **split**. It combines its arguments into a single string and uses the delimiter as the first argument. The remaining arguments could be either an array name or a list of variables or strings to be joined. This is how you provide a space after each day:

```
$weekstring = join(" ", @week_array) ;
$weekstring = join(" ", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun") ;
print $weekstring ;
```

Either statement should produce this output:

```
Mon Tue Wed Thu Fri Sat Sun
```

split and **join** often go together. The next program, **rep2.pl** (Fig. 19.9), splits each line of our sample database on the |, adds a century prefix to the date and then joins all fields back. The script is well documented to need elaboration.

Let's now print the first two lines of our transformed database by running the program:

```
$ rep2.pl emp.lst | head -n 2
2233|a.k. shukla      |g.m.     |sales      |12/12/1952|6000
9876|jai sharma       |director |production|12/03/1950|7000
```

```
#!/usr/bin/perl -n
# Script: rep2.pl - Uppercases the name and adds century prefix to the date
#
@line = split(/\|/) ;                          # $_ is assumed
($day, $month, $year) = split(/\//, $line[4]);  # Splits date field
$year = "19" . $year ;                         # Adds century prefix
$line[4] = join("\/", $day, $month, $year);    # Rebuilds date field
$line = join("\|", @line);                     # Rebuilds line
print $line;
```

Fig. 19.9 rep2.pl

Joining on a specified delimiter has common applications in everyday programming. Even though we used **join** on a specific delimiter in our examples, the next section uses **join** without any delimiter to perform a very useful task.

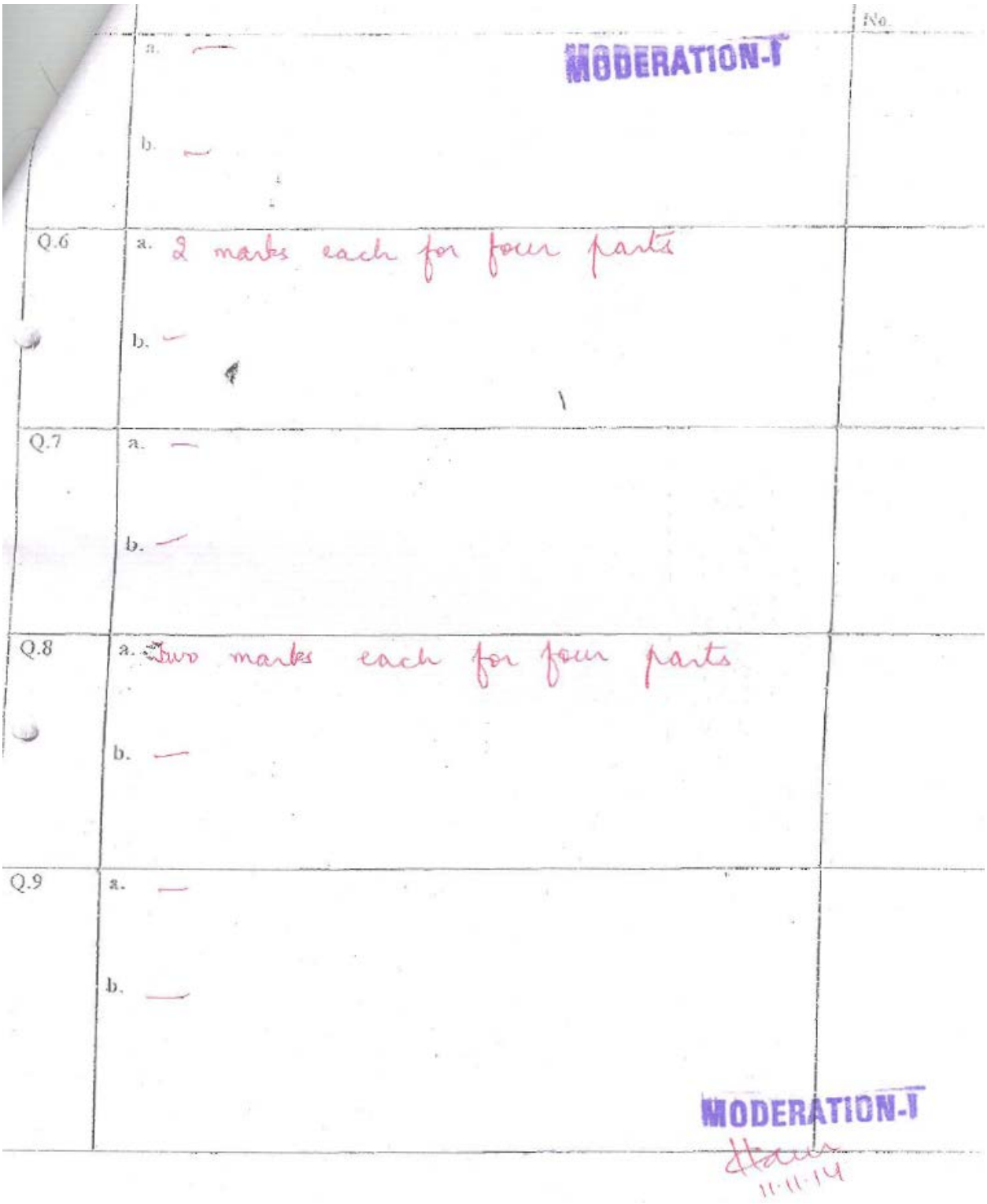
CODE AC 109/AT 109 SUBJECT Unix & Shell Programming

(Marking Scheme)

Marking scheme mentioned in the question paper itself for rest of the questions. Hari 11.11.14

Q.		Unit, Text Book, Page Contents No
Q.2	a. — b. —	
Q.3	a. — b. 2 marks each for all four parts.	
Q.4	a. — b. 2 marks each for all four parts	

MODERATION-I



TEXT BOOK

I. UNIX Concepts and Applications, 4th Edition, Sumitabha Das, Tata McGraw Hill, 2008