CMR Institute of Technology

Unix and Shell Programming-18MCA12

Answer Key-December 2019

1.a. Explain the UNIX architecture with a neat diagram.

UNIX OS distributes its major jobs into two agencies viz. kernel and shell.

Kernel: The kernel is also known as operating system. It interacts with the hardware of the machine. The kernel is the core of OS and it is a collection of routines/functions written in C. Kernel is loaded into memory when the system is booted and communicates with the hardware. The application programs access the kernel through a set of functions called as system calls. The kernel manages various OS tasks like memory management, process scheduling, deciding job priorities etc. Even if none of the user programs are running, kernel will be working in a background.

Shell: The shell interacts with the user. It acts as a command interpreter to translate user's command into action. It is actually an interface between user and the kernel. Even though there will be only one kernel running, multiple shells will be active – one for each user. When a command is given as input through the keyboard, the shell examines the command and simplifies it and then communicates with the kernel to see that the command is executed. The shell is represented by sh (Bourne Shell), csh (C Shell), ksh (Korn shell), bash (Bash shell).

The relationship between kernel and shell is shown in Figure

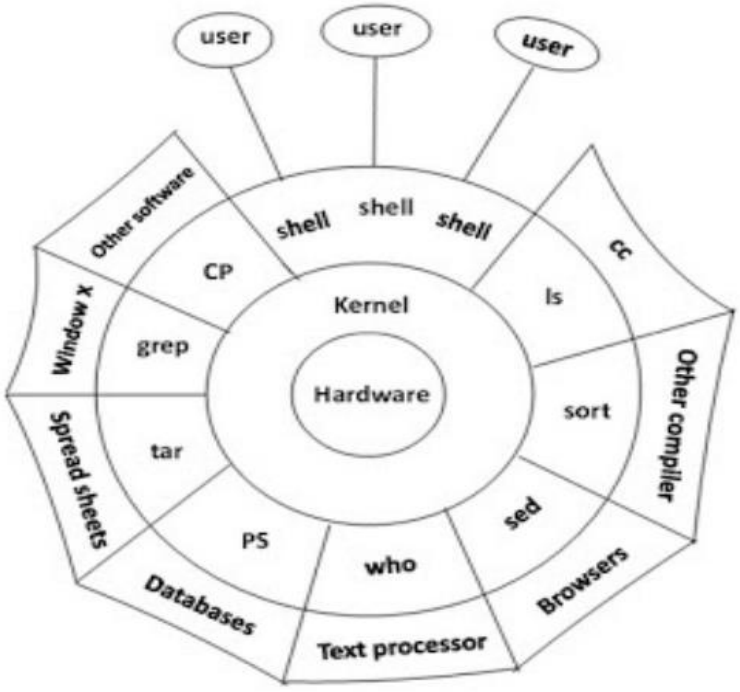


Figure 1.1 The Kernel – Shell Relationship

The File and Process

The file and the process are two simple entities that support UNIX.

File: A file is an array of bytes and it can contain any data. Various files are related to each other by a hierarchical structure. Even a user (user name) is placed in this file system. UNIX considers directories and the devices also as the members of file system. In UNIX, the major file type is text and the behavior of UNIX is controlled mainly by text files. UNIX provides various text manipulation tools through which the files can be edited without using an editor.

Process: A processing a program under execution. Processes are also belonging to separate hierarchical structure. A process can be created and destroyed. UNIX provides tools to the user to control the processes, move them between foreground and background and to kill them.

The System Calls:

System calls are used to communicate with the kernel. There are more than thousand commands in UNIX, but they all use few set of function called as system calls for communication with kernel. All UNIX flavors (like Linux, Ubuntu etc) all use the same system calls. For example, write is a system call in UNIX. C programmer in UNIX environment can directly use this system call to write data into a file. Whereas, the C Programmer in Windows environment may need to use library function like fprintf() to write into a file. A system call open in UNIX can be used to open a file or a device. Here, the purpose is different, but the system call will be same. Such feature of UNIX allows it to have many commands for user purpose, but only few system calls internally for the actual work to be carried out in association with the kernel.

1.b. Explain in detail any five feature of UNIX operating system.

• /var/tmp : A place for temporary files which should be preserved between system reboots.

A Multiuser System: UNIX is basically a multiprogramming system. Here, either Multiple users can run separate jobs or Singe user can run multiple jobs. In UNIX, many processes are running simultaneously. And, the resources like CPU, memory and hard disk etc are shared between all users. Hence, UNIX is a multiuser system as well. The Unix system breaks up one time unit into several segments and each user is allotted one segment. At any point of time, the machine will be doing the job of one user. When the allotted time expires, the job is temporarily suspended and next user's job is taken up. This process continues till all processes gets one segment each and once again the first user's job is taken up. The kernel does this task several times in one second such a way that the users will never come to know about it and users cannot make out the delay in between.

A Multitasking System: Unix is a multitasking system, wherein a single user can run multiple jobs concurrently. A user may edit a file, print a document on a printer and open a browser etc – all at a time. In multitasking environment, a user can see one job running in the foreground and all other jobs run in the background. The jobs can be switched between background and foreground; they can be suspended or terminated.

The Building-Block Approach: Unix is a collection of few hundred commands, each of which is designed to perform one task. More than one command can be connected via the | (pipe) symbol to perform multiple tasks. The commands which can be connected are called as filters because, they filter or manipulate data in different ways. Many Unix tools are designed such a way that the output of one tool can be used as input to another tool. For this reason, UNIX commands do not generate lengthy or messy outputs. If a program is interactive, then user's response to it may be different. In such situations, the output of one command cannot be made as input to another command. Hence, UNIX programs are not interactive.

The UNIX Toolkit: Unix contains diverse set of applications like text manipulation utilities, compilers and interpreters, networked applications, system administration tools etc. The Unix kernel does many tasks with the help of these applications. Such set of tools are constantly varying with every release of UNIX.

In every release, new tools are being added and old tools are either removed or modified. Most of these tools are open-source utilities and the user can download them and configure to run on one's machine.

Pattern Matching: Unix has very sophisticated pattern matching features. The character like * (known as a metacharacter) helps in searching many files starting with a particular name. Various characters from a metacharacter set of Unix will help the user in writing regular expressions that will help in pattern matching.

Programming Facility: The Unix shell is a programming language as well. It provides the user to write his/her own programs using control structures, loops, variables etc. Such programs are called as shell scripts. Shell scripts can invoke Unix commands and they can control various functionalities of Unix OS.

Documentation: Unix provides a large set of documents to understand the working of every command and feature of it. The man command can be used on an editor to get the manual about any Unix command. Moreover, there are plenty of documents, newsgroups, forums and FAQ (Frequently Asked Questions) files available on internet, where one can get any information about Unix.

2.a. Explain the following commands.

i)cat ii)bc iii)date iv)script v)tty

i)cat

It is a short-form of the term concatenate. This command is basically used for viewing the contents of a file. But, it has many other usages like creating a file, joining more than one file etc. Here, few of the usages of cat command are discussed.

 To create a new file:

Following is an example to create a new file –

```
$ cat >test hello how are you?
   I'm doing good.
what about you?
  [Ctrl+d] $
```

To display contents of a file: The cat command is used with filename to display the contents of the file as shown below –

```
$cat t1
This is first file          $
```

bc

UNIX provides two types of calculators – a graphical (GUI) calculator (similar to the one available in windows OS) and a character based ***bc*** command. A visual calculator can be available using ***xcalc*** command and it is available only on X Window system, but not on command-line based terminals.

The calculator available through ***bc*** command is a very powerful, but sadly a most neglected tool in UNIX. When ***bc*** command is invoked without any argument, it does nothing but waits for the input from the keyboard. Once the job is done, *ctrl+d* has to be pressed to release the command and to get a prompt.

The usage of ***bc*** command is illustrated here with examples.

- **Basic operations:**

```
$bc
3+5
8
5*6
30
6-10
```

```
-4
[ctrl+d]
```

- **To perform more than one operation in a single line:**

```
$bc
2^4; 3+6 //using semicolon as a separator
16
9
[ctrl+d]
```

- **Setting scale for required precision during division operation:**

By default, ***bc*** performs truncated division (or integer division). For example,

```
$bc
9/5
1
```

Here, the output 1, instead of 1.8. To avoid such truncation, one can set the precision after the decimal point. For example,

```
$bc
scale=2
9/5
1.80
22/7
3.14
```

- **Converting numbers from one base to the other:**

One can change the base of a number by setting ***ibase*** (input base) or ***obase*** (output base). For example –

```
$bc
ibase=2 //setting input base as 2
1100
12 //decimal equivalent of 1100
11001110
206
```

ii)date

The ***date*** command in UNIX is used to display the current date and time of the system. The UNIX system maintains an internal clock that runs continuously. When the system is shutdown, the battery backup keeps the clock ticking. This clock actually stores the number of seconds elapsed since 1st January 1970. A 32-bit counter store these seconds and it is expected to overflow sometime in 2018.

The format of the *date* command is "Day Month date hr:min:sec IST year". For example –

$date

Mon Jun 30 11:35:32 IST 2017

Suitable format specifiers can be used as an argument to *date* command to get the date/month/year etc. in required format. The format specifier is preceded by a + symbol, followed by the % operator and a single character describing the format. Following are some of the formats –

 To display only the month, one can use +%m as a specifier. For example,

$date +%m

10 //indicates October.

 To display month name, use +%h as below –

$date +%h

Oct

 +%d for day of month (1 – 31)
+%y for last two digits of the year
 +%H, +%M and +%S indicates hour, minute and second respectively.
 One can combine more than one option by enclosing them in double quotes, and keeping + symbol outside the quote. For example, combination of month and month name –

$date +"%h %m"
Oct 10

script:

The ***script*** command is used to record the session in a file. When you have are doing some important work, and would like to keep a log of all your activities, you should use ***script*** command immediately after logging in. For example,

$script
Script started, file is typescript
$

Now onwards, whatever you type, that will be stored in the file *typescript*. Once the recording is over, you can terminate the session by using ***exit*** command.

$exit
Script done, file is typescript
$

To view the file *typescript*, one can use ***cat*** command.

Note that, the usage of *script* command overwrites any existing file with name *typescript.* If you want to append the new content to existing file, then use ***–a*** as below –

$script –a

Now, the previous *typescript* will be appended with the activities of this session.

If you want to create your own file instead of *typescript* file, then give the required filename as –

$script mylogfile

Now, the activities of this session will be stored in the file *mylogfile.*

NOTE that, some activities like the commands used in the full-screen mode like ***vi*** editor will not be recorded properly when we record session using *script* command.

tty:

The command *tty* (teletype) is used to know name of the terminal. For example,

```
$tty
/dev/tty01
```

The above statement indicates that *tty01* is the name of the terminal and it is within the directory *dev.* The *dev* is under *root* directory.

NOTE that, UNIX treats all devices as files, and *tty01* is one of the files under file system. In UNIX, just like users, even terminals, disks and printers also have the name and all these are treated as files. Even the commands are also files in UNIX.

Some the systems may display the statement as below when *tty* command is given –

```
$tty
/dev/pts/1
```

Here, pts/1 is the name of the terminal.

2.b. Explain the following in detail with examples.

i)read ii)command line argument iii)exit status of a command.

i)read

The read statement is the internal tool of the shell for taking input from the user. This will help the scripts to become interactive.

example –

read fname lname

ii)command line arguments

Arguments can be passed to a shell script through the command line, while running the script. Such command line arguments are assigned to special variables known as positional parameters. The name of the program itself is treated as first argument and stored in positional parameter $0. Further arguments given by the user are sequentially stored in parameters $1, $2 and so on. Note that, these are not called as shell variables (because, name of shell variables starts with a character). Some of the special parameters used by the shell are listed in Table

Table 1.1 Special parameters used by Shell

Shell Parameter	Significance
$0	Name of the executed command (that is script file name – in most of the cases)
$1, $2 etc.	Positional Parameters representing command line arguments
$#	Total number of arguments specified in command line (excluding file name)
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as a separate argument
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

iii)exit status of the command

Whenever a shell command gets executed, its execution status is stored in a special parameter $?. If the command is executed successfully, then 0 will be stored (indicating true). Otherwise, some non-zero number (indicating false) is stored in $?. It is called as exist status of a command. This parameter always contains the exit status of the last command which has been executed. Consider few examples:

Ex1: Successful Execution

$echo "Date: `date`" ;

echo "Exit Status: $?"

Date: Wed Oct 18 21:25:29 IST 2017 Exit Status: 0

Here, the date command is executed within echo. As, it could execute successfully, the exit status parameter will be obviously 0.

3.a. Explain the following commands.

i)pwd ii)cd iii)mkdir iv)rmdir

i)pwd

Once a user logs in to the UNIX system, it places him in a specific directory (usually home directory) of the file system. Though a user can move from one directory to other, for a given moment of time, he will be in one directory, known as current directory. To know current directory, the pwd (print working directory) command is used. The pwd command displays the absolute pathname as below –

$pwd

/home/john

ii)cd

The cd command is used to move around the file system by changing the directory. This command can be used in three different ways – If the user john is in his home directory and would like to move to

subdirectory called as progs, then the command should be given as – $ cd progs # user is moved to progs directory now $ pwd # verify this using pwd /home/john/progs

mkdir:

This is the command used to create a new directory. Directories can be used to have a collection of files under a common name.

$mkdir docs

$_

The *mkdir* command has created directory viz. *docs*. But, the command has not shown any output, but just displays the prompt. Many a times, UNIX commands do not display any output, but does the job internally. Here, *mkdir* has created the directory and that can be confirmed using *ls* command.

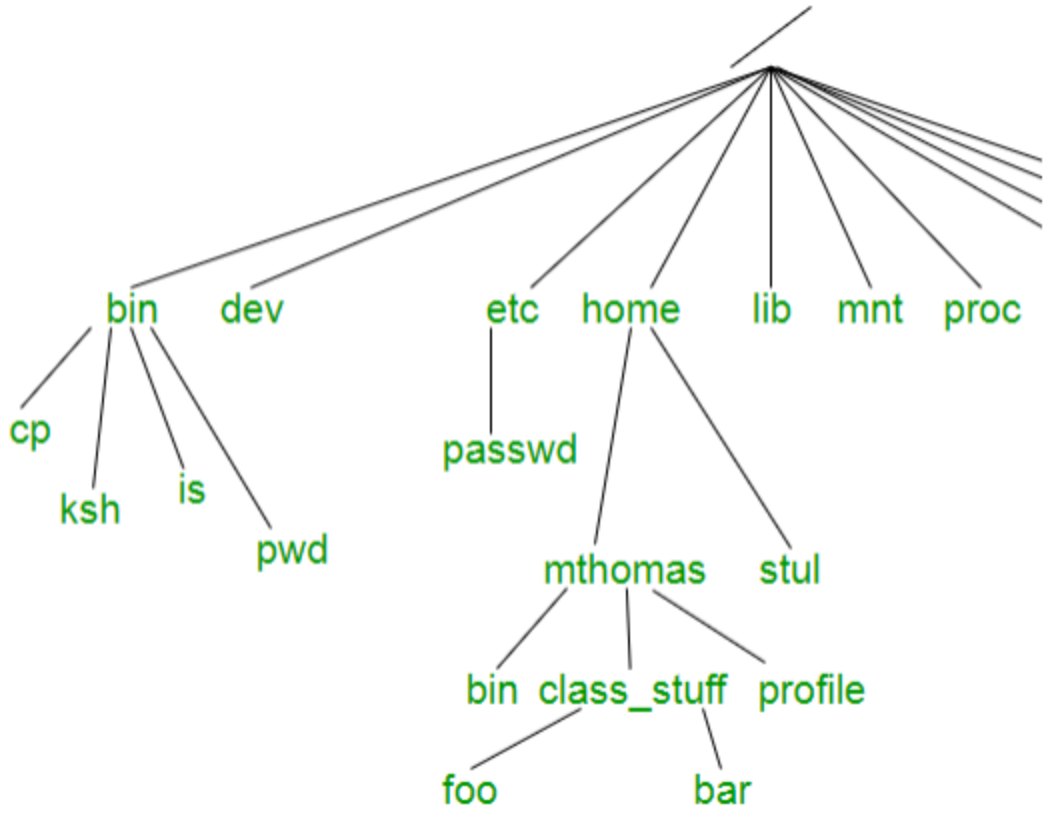
Note that, UNIX internally treats a directory also as a file.

iv)rmdir

To remove (or delete) a directory, the rmdir command is used. Few important points about this command are discussed here – A directory has to be empty before removing it. That is, it should not contain any files or subdirectories. To remove one directory, use statement like – $rmdir test #removes the directory test

3.b. Explain the UNIX file system with a neat diagram.

Unix file system is a logical method of organizing and storing large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system. Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called “root” which is represented by a “/”. All other files are “descendants” of root.



Directories or Files and their description –

• / : The slash / character alone denotes the root of the filesystem tree.

• /bin : Stands for “binaries” and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.

• /boot : Contains all the files that are required for successful booting process.

• /dev : Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices.

• /etc : Contains system-wide configuration files and system databases. Originally also contained “dangerous maintenance utilities” such as init, but these have typically been moved to /sbin or elsewhere.

• /home : Contains the home directories for the users.

• /lib : Contains system libraries, and some critical files such as kernel modules or device drivers.

• /media : Default mount point for removable devices, such as USB sticks, media players, etc.

• /mnt : Stands for “mount”. Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.

• /proc : procfs virtual filesystem showing information about processes as files.

• /root : The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial file system, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed,
during which other filesystems are not available. Such a case could occur,
for example, if a hard disk drive suffers physical failures and cannot be properly mounted.

• /tmp : A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
• /usr : Originally the directory holding user home directories,its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).
• /usr/bin : This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.
• /usr/include : Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language.
• /usr/lib : Stores the required libraries and data files for programs stored within /usr or elsewhere.
• /var : A short for "variable." A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
• /var/log : Contains system log files.
• /var/mail : The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.
• /var/spool : Spool directory. Contains print jobs, mail spools and other queued tasks.

4.a. Describe the seven attributes of the ls –l command.

Output in Multiple Columns (–x) : When there are many files, it is better to display them in multiple columns. Modern versions of ls do that by default (without any options), but if it doesn't happen in your system, you can use –x option as – $ ls –x Thesis Shell1.sh Shell2.sh ShellPgms Emp.txt cmd.c helpdir

 Identifying Directories and Executables (–F) : The ls command displays files as well as directories. To know which of them are directories and executable files, one can use –F option. When –F option is combined with –x, it produces multicolor output.

$ ls –Fx

Thesis/ Shell1.sh* Shell2.sh* ShellPgms/ Emp.txt cmd.c helpdir/

Here, the symbols * and / are type indicators. The * indicates that the file contains executable code, and / refers to a directory.

 Showing Hidden Files Also (–a): If we want to see hidden files also, we use –a (all) option for ls. There are certain hidden files starting with a dot, which normally don't get displayed with just ls command.

$ ls –a

. .. .exrc Thesis .emacs .gnome2 Shell1.sh

Note that, the first two files displayed are . and .. indicating current and parent directories.

 Listing specific directory contents: If you want to display the contents of only specific subdirectories, you can give the name along with ls as shown below –

$ ls ACMPaper ShellPgms

ACMPaper: acm.aux acm.bib acm.pdf runtex

acm.tex sig.pdf

ShellPgms:

caseEx.sh first.sh menu.sh test

hello.c

 Recursive Listing (–R): This option lists all files and subdirectories in a directory tree. That is, contents of subdirectories also will be displayed recursively till there is no subdirectory is left out.

```
$ ls –R .:
Thesis
 Shell1.sh  Shell2.sh
 ShellPgms
 Emp.txt cmd.c
 ./Thesis: Chap1.aux  Chap1.bib  Chap1.tex  Chap1.pdf Annex.aux  Annex.pdf
 ./ShellPgms: caseEx.sh  first.sh  menu.sh test hello.c
 ./ShellPgms/helpdir: Test.c  here.sh  try.sh
```

One can observe that, the –R option starts display with the current directory (.). Then it displays the contents of all subdirectories under current directory. Later it goes one level down and so on.

ls –l option:

Listing File Attributes The –l option of ls command is used for listing the various attributes like permissions, size, ownership etc. of a file. The output of ls –l is referred to as the listing. The –l option can be combined with other options for displaying other attributes, or ordering the list in a different sequence. The command ls use inode of a file to fetch its attributes. Consider the following example of ls –l which displays seven attributes of all files in the current directory.

```
 $ ls -l                    total 144                    -rw-rw-r-- 1 john john
280 Jan 30 09:56 caseEx.sh       -rw-rw-r-- 1 john john  104 Feb  3 06:40 cmdArg.sh
```

ls –d option: Listing Directory Attributes If we want to list the attributes of only the directory, but not its contents, we can use –d option as below –

```
 $ ls –ld myDir
 drwxrwxr-x 2 john john 4096 Feb  6 05:48 myDir
```

4.b Explain about hard link and symbolic link with examples.

Hard Link:

A link can be created to a file using ln command. The following command is used to create hard link for an existing file emp.lst with a non-existing file employee:

```
  $ln emp.lst employee
$ ls –li emp.lst employee
 29314 –rwxr-xr-x 2 john metal 915 May 5 03:34 emp.lst
 29314 lrwxr-xr-x 2 john metal 915 May 5 03.34 employee
```

The hard links has certain limitations – One cannot have two linked filenames in two file systems. That is, one cannot link a filename in the /usr file system to another in /home file system. One cannot link a directory even within the same file system.

The symbolic links or soft links will overcome these limitations. The symbolic link can be thought of as a fourth type of a file (apart from 3 types discussed till now – ordinary, directory and device). Unlike the hard link, a symbolic link doesn't have the file's contents. But, it simply provides the pathname of the file that actually has the contents. Shortcut keys in windows are the best examples for symbolic links.

The ln command with –s option is used to create symbolic link as below –

```
 $ ln –s note note.sym
 $ ls –li note note.sym
 9948 –rw-r--r-- 1 john metal 915 May 5 03:34 note
 9952 lrwxrwxrwx 1 john metal   4 May 5 03:34 note.sym->note
```

5.a. Explain the following commands.

i)head ii)tail iii)pr iv)cut v)paste

i)head

The head command is used to display top of the file. By default, it displays first 10 lines of the file. The -n option can be used with a required line–count to display those many lines. For example,

```
$ head -n 3 emp.lst
  2233|a.k. shukla|g.m.|sales|12/12/52|6000
 9876|jai sharma|director|production|03/12/50|7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

Here, -n is used with the count 3; hence the top 3 lines of the file will be displayed.

ii)tail

The tail command is opposite to head. That is, it displays last 10 lines by default. By specifying the count with –n option, we can display only the required number of lines as below –

```
 $ tail -n 3 emp.lst
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

ii)pr

The pr command prepares a file for printing by adding suitable headers, footers, and formatted text. When used with a filename as argument, pr doesn't behave like a filter:

```
$ pr group1
May 06 10:38 1999 group1 Page 1
root:x:0:root These seven lines are the original
bin:x:1:root,bin,daemon contents of group1
users:x:200:henry,image,enquiry
adm:x:25:adm,daemon,listen
dialout:x:18:root,henry
lp:x:19:lp
ftp:x:50:
... blank lines ...
```

pr adds five lines of margin at the top (simplified here) and five at the bottom. The header shows the date and time of last modification of the file, along with the filename and page number. We generally don't use pr like this. Rather, we use it as a "preprocessor" to impart cosmetic touches to text files before they are sent to the printer:

```
$ pr group1 | lp
Request id is 334
```

Since pr output often lands up in the hard copy, pr and lp form a common pipeline sequence. Sometimes, lp itself uses pr to format the output, in which case this piping is not required.

iii)Cut

Cutting Columns (-c) To extract specific columns, you need to follow the -c option with a list of column numbers, delimited by a comma. Ranges can also be specified using the hyphen. Here's how we extract the first four columns of the group file:

```
$ cut -c1-4 group1 -c or -f option always required
root
bin:
user
adm:
dial
lp:x
```

ftp:

v)paste

The paste command is used to paste the contents into a file vertically. One can view two files side by side by pasting them. To understand the working of paste command, first let us create two files by cutting some fields from emp.lst.

The file cutlist1 is created as below, which contains 2nd and 3rd fields (name and designation) of the file emp.lst.

```
$ cut -d \| -f 2,3 emp.lst > cutlist1
 $ cat cutlist1
  a.k. shukla|g.m.                    jai sharma|director                    sumit
chakrobarty|d.g.m                   barun sengupta|director
```

5.b. Explain about regular expression in detail. What are the various wild card characters used in regular expressions.

A regular expression (regex) is defined as a pattern that defines a class of strings. Given a string, we can then test if the string belongs to this class of patterns. Regular expressions are used by many of the UNIX utilities like *grep*, *sed*, *awk*, *vi* etc. A regular expression is a set of characters that specify a pattern. Regular expressions are used when we want to search for specify lines of text containing a particular pattern. Regular expressions search for patterns on a single line, and not for patterns that start on one line and end on another.

The regular expression uses meta-character set as given in Table 3.4.

Table 3.4 Character subset for BRE

Symbol/ Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg etc
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character *p*, *q* or r
[c1-c2]	A single character within the ASCII range represented by c1 and c2
[1-3]	A digit between 1 and 3
[^pqr]	A single character which is not *p*, *q* or *r*
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern *pat* at the beginning of line
pat$	Pattern *pat* at end of line
^pat$	*pat* as only word in line
^$	Lines containing nothing

The Character Class

A regular expression lets to specify a group of characters enclosed within a pair of rectangular brackets [ ]. The match is performed for a single character in the group. For example, the expression [ra] matches either r or a.

In the previous section, we have seen that ***grep*** with –e option is used to compare multiple patterns. Now, let us write the regular expression for searching different spellings of *agarwal* in *emp.lst*.

```
$ grep "[aA]g[ar][ar]wal" emp.lst
```

3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000

The *

The * refers to the *immediately preceding* character. It matches *zero or more occurrences of the previous character*. Hence, the pattern g* matches *null string* or following strings –
g, gg, ggg, gggg ……..
As the * can match even a null string, if you want to search a string beginning with *g*, do not give pattern as g* , instead give as gg*.
Now check the following example, where all three types of spellings of *agarwal* can be searched.

$ grep "[aA]gg*[ar][ar]wal" emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000

The Dot (.)

The dot (.) matches a single character. For example, the pattern 2… matches a fourcharacter pattern beginning with a 2. The combination of * and dot (.*) constitutes a very useful regular expression. It signifies any number of characters or none. For example, when you are not sure about the initial of *saxena,* you can give the expression as -

$ grep ".*saxena" emp.lst
2345|j.b. saxena|g.m.|marketing|03/12/45|8000

Specifying Pattern Locations (^ and $)

When we need to search for a pattern either at the beginning or at the end of a line, we can use ^ and $ respectively. For example, following command searches all the employees whose employee ID starts with 2.

$ grep "^2" emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
2365|barun sengupta|director|personnel|05/11/47|7800
2476|anil aggarwal|manager|sales|05/01/59|5000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000

6.a. Describe about grep command in detail with any five options used in it.

The ***grep*** command scans its input for a pattern and displays lines contain the pattern, the line numbers or filenames where the pattern occurs. The syntax is –
grep *options pattern filename(s)*
Various options for ***grep*** command are given in Table 3.3. They are discussed with suitable examples below.
Table 3.3 Options for *grep* command

Table 3.3 Options for *grep* command

Option	Description
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e *exp*	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep –** style)

 Ignoring Case (-i): When we are searching for a pattern, but not sure about the case, -i option is used. It ignores the case of the text and displays the result. For example,

$ grep -i 'agarwal' emp.lst

3564|sudhir Agarwal|executive|personnel|07/06/47|8000

 Deleting Lines (-v): The –v (inverse) option selects all lines except those containing the pattern. The following example selects all lines in the file *emp.lst* except for those containing the term *director*.

$ grep -v 'director' emp.lst

2233|a.k. shukla|g.m.|sales|12/12/52|6000

5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000

5423|n.k. gupta|chairman|admin|08/30/56|5400

6213|karuna ganguly|g.m.|accounts|06/05/62|6300

Displaying Line Numbers (-n): This option displays the line numbers containing the pattern along with the actual lines. For example,

$ grep -n 'marketing' emp.lst

3:5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000

11:6521|lalit chowdury|director|marketing|09/26/45|8200

14:2345|j.b. saxena|g.m.|marketing|03/12/45|8000

15:0110|v.k. agrawal |g.m.|marketing|12/31/40|9000

 Counting Lines Containing Pattern (-c): A pattern may be present in a file multiple times. If we would like to know how many times it has appeared, -c option can be used. The following example shows how many times the pattern *director* has appeared in the file *emp.lst*.

$ grep -c 'director' emp.lst

4

 Displaying Filenames (-l): The –l (el) option is used to display the names of files containing the pattern. Assume there are there are two more files *test.lst* and *testfile.lst* along with *emp.lst*. Now, let us check in which file(s) the pattern *manager* is present.

$grep –l 'manager' *.lst

emp.lst

test.lst

 Matching Multiple Patterns (-e): When we would like to search for multiple patterns in a file, we can use –e option. For example,

$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
 Taking Patterns from a File (-f): If various patterns are stored in a file each in different line, then –f option can be used by giving that filename as one of the arguments. For example, assume there is a file *pattern.lst* as –
$cat >pattern.lst
manager
executive
Then, give the command as –
$grep –f pattern.lst emp.lst
1265|s.n. dasgupta|manager|sales|09/12/63|5600
4290|jayant Chodhury|executive|production|09/07/50|6000
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000

6.b. Write about line addressing and context addressing with examples.
The ***sed*** command is a multipurpose tool which combines the work of several filters. It performs non-interactive operations on a data stream. It allows selecting lines and running instructions on them.
An instruction combines an ***address*** for selecting lines, with an ***action*** to be taken on them. The ***sed*** command uses such instructions. The syntax is –
sed *options 'address action' file(s)*
Line Addressing: Here, *address* specifies either one line number to select a single line or a set of two numbers to select a group of contiguous lines.

Option	Description
`d`	Deletes line(s)
`10q`	Quits after reading the first 10 lines
`p`	Prints line(s) on standard output
`3,$p`	Prints lines 3 to end (-n option required)
`$!p`	Prints all lines except last line (-n option required)
`/begin/, /end/p`	Prints lines enclosed between *begin* and *end* (-n option required)
`q`	Quits after reading up to addressed line
`r flname`	Places contents of file *flname* after line
`w flname`	Writes addressed lines to file *flname*
`=`	Prints line number addressed.

In line addressing, the instruction *3q* can be broken into the address 3 and the action q (quit). So, to display only first 3 lines, (similar to *head –n 3*) use the following statement –
$ sed '3q' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
In the above example, 3 lines will be displayed and then quits.
Generally, the ***p*** (print) command is used to display lines. But, this command behaves

strange – it prints selected lines as well as *all* lines. Hence, the selected lines will appear twice. To suppress this feature of *p*, the –n option has to be used. The following example selects the lines 5 through 7.

```
$ sed -n '5,7p' emp.lst
5423|n.k. gupta|chairman|admin|08/30/56|5400
1006|chanchal sanghvi|director|sales|09/03/38|6700
6213|karuna ganguly|g.m.|accounts|06/05/62|6300
```

The $ symbol can be used to print only the last line as below –

```
$ sed -n '$p' emp.lst
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

The ***sed*** command can be used to select multiple groups of lines. In that case, each address has to be given in a different line, but enclosed within a single pair of quotes as shown below –

```
$ sed -n '1,2p
> 7,9p
> $p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
6213|karuna ganguly|g.m.|accounts|06/05/62|6300
```

The ***sed*** command uses ! (exclamatory mark) as a negation operator. Assume, we would like to select first 2 lines of the file. Note that, selecting first two lines means – not selecting 3rd line to end. So, the command can be used as below –

```
$ sed -n '3,$!p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

Here, ! is for ***p*** indicating not to print lines from 3 to end.

Using Multiple Instructions (-e and –f) : In the previous section, we have seen that when multiple groups of lines have to be selected, the pattern should be given in different lines with a line-break in-between. To avoid that, ***sed*** uses –e option. This option allows to enter as many instructions as you wish, in a single line, where each instruction is preceded by the option –e. For example, the following command selects multiple lines (1 to 2, 7 to 9 and last line) –

```
$ sed -n -e '1,2p' -e '7,9p' -e '$p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

When we have too many instructions to use or when we have a set of a common instructions that are executed often, better to store them in a file. And, then use –f option with ***sed*** command to read from that file and to apply the instructions on input file. Consider the example given below. Here, we have created a file *instr.dat* containing required instructions. Then use the ***sed*** command.

```
$ cat >instr.dat
1,2p
7,9p
$p
$ sed –n –f instr.dat emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

Context Addressing:

Context addressing allows to specify one or two patterns to locate the lines. The patterns must be bounded by a / on both the sides. When a single pattern is specified, all lines containing the pattern are selected. The following example is for selecting all the lines containing the pattern *director*.

```
$ sed -n '/director/p' emp.lst
9876|jai sharma|director|production|03/12/50|7000
2365|barun sengupta|director|personnel|05/11/47|7800
1006|chanchal sanghvi|director|sales|09/03/38|6700
6521|lalit chowdury|director|marketing|09/26/45|8200
```

One can give a comma-separated list of context addresses to select a group of lines. For example, to select all the lines between *dasgupta* and *saxena* use the following statement –

```
$ sed -n '/dasgupta/,/saksena/p' emp.lst
1265|s.n. dasgupta|manager|sales|09/12/63|5600
4290|jayant Chodhury|executive|production|09/07/50|6000
2476|anil aggarwal|manager|sales|05/01/59|5000
```

One can mix line addressing and context addressing. If we want to select all lines from 1st line till *dasgupta*, use the command as below –

```
$ sed -n '1,/dasgupta/p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

Regular expressions can be used as a part of context address. For example, the following command selects different spellings of *agarwal.*

```
$ sed -n '/[aA]gg*[ar][ar]wal/p' emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

One more example of ***sed*** command including regular expression is given below. It selects all lines containing *saksena, saxena,* and *gupta*. Note that, here also two different patterns should be given on different lines.

```
$ sed -n '/sa[kx]s*ena/p
> /gupta/p' emp.lst
2365|barun sengupta|director|personnel|05/11/47|7800
5423|n.k. gupta|chairman|admin|08/30/56|5400
1265|s.n. dasgupta|manager|sales|09/12/63|5600
```

The characters ^ and $ also can be used as a part of regular expression with ***sed*** command. Following example shows the people born in 1950. Note that, the five dots after 50 in the expressions indicate 5 characters (a delimiter | and 4 characters indicating salary) present before the end of line ($).

```
$ sed -n '/50.....$/p' emp.lst
9876|jai sharma|director|production|03/12/50|7000
4290|jayant Chodhury|executive|production|09/07/50|6000
```

7.a. Explain about awk-filter with syntax and example.

The syntax of *awk* command is –

awk *options 'selection_criteria {action}' file(s)*

Here, the *selection_criteria* is a form of addressing and it filters input and selects lines. The *action* indicates the action to be taken upon selected lines, which must be enclosed within

the flower brackets. The *selection_criteria* and *action* together constitute an ***awk*** program enclosed within single quotes. The *selection_criteria* in *awk* can be a pattern as used in context addressing. It can also be a line addressing, which uses ***awk'***s built-in variable **NR.** The *selection_criteria* can be even a conditional expressions using && and || operators. Consider the following example of ***awk*** command to select all the *directors* in the file *emp.lst.*

```
$ awk '/director/{print}' emp.lst
9876|jai sharma|director|production|03/12/50|7000
2365|barun sengupta|director|personnel|05/11/47|7800
1006|chanchal sanghvi|director|sales|09/03/38|6700
6521|lalit chowdury|director|marketing|09/26/45|8200
```

Here, the selection criteria is /director/which selects the lines containing *director*. The action is {print}. If the selection criteria is missing, then *action* applies to all the lines. If *action* is missing, then entire line is printed.

The ***print*** statement prints the entire line, when it is used without any field specifier. It is a default action of ***awk.***

7.b. What are the various built-in operators used in awk? Explain.

The Comparison Operator

The relational operators like greater than (>), less than (<), comparison (==), not equal to (!=) etc. can be used with *awk*.

Ex1. The command given below is to select the lines containing *director* OR (||) *chairman*. As the 3rd field of the file *emp.lst* has the designation, we can compare it directly.

```
$ awk -F "|" '$3=="director" || $3=="chairman" {
> printf "%-20s %-12s \n", $2, $3}' emp.lst
jai sharma director
barun sengupta director
n.k. gupta chairman
chanchal sanghvi director
lalit chowdury director
```

Ex2. Using not equal to operator (!=) and AND (&&) operator, we can achieve the negation of list obtained in Ex1. That is, following command displays all the lines not containing *director* and *chairman*.

```
$ awk -F "|" '$3 != "director" && $3 != "chairman" {
>printf "%-20s %-12s \n", $2, $3}' emp.lst
a.k. shukla g.m.
sumit chakrobarty d.g.m
karuna ganguly g.m.
s.n. dasgupta manager
jayant Chodhury executive
anil aggarwal manager
shyam saksena d.g.m
sudhir Agarwal executive
j.b. saxena g.m.
v.k. agrawal g.m.
```

The Regular Expression Operators: ~ and !~

The *awk* provides two operators for handling regular expressions. The ~ operator is used to match a regular expression and !~ is used to negate the match. These operators must be used only with field specifiers like $1, $2 etc.

Ex1. In the below example, the 2nd field ($2) is matched with regular expressions that may result in any of *Chodhury*, *chowdury, saksena, saxena*. Observe the OR (||) operator used.

```
$ awk -F "|" '$2 ~ /[cC]how*dh*ury/ ||
> $2 ~ /sa[kx]s?ena/ {print}' emp.lst
4290|jayant Chodhury|executive|production|09/07/50|6000
6521|lalit chowdury|director|marketing|09/26/45|8200
3212|shyam saksena|d.g.m|accounts|12/12/55|6000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

Number Comparison

awk can handle both integer and floating type numbers. Relational operators also can be applied on them. Consider following examples:

Ex1. Listing of all the employees whose salary is more than 7500 can be done as below –

```
$ awk -F"|" '$6>7500 {printf "%-20s %d\n",$2,$6}' emp.lst
barun sengupta 7800
lalit chowdury 8200
sudhir Agarwal 8000
j.b. saxena 8000
v.k. agrawal 9000
```

Here, $6 is the 6th field (salary), is being compared with 7500.

Ex2. One can combine regular expression matching and numeric comparison. Following example lists out the people who have either born in 1945 OR getting the salary more than 8000.

```
$ awk -F"|" '$6>8000 || $5~/45$/{print $2, $5, $6}' emp.lst
lalit chowdury 09/26/45 8200
j.b. saxena 03/12/45 8000
v.k. agrawal 12/31/40 9000
```

In the file *emp.lst,* 5th field is date of birth. In this field, at the end we have year of birth. Hence, in the above example, $5 is matched with /45$/ indicated 45 is at the end (recollect meaning of $ in regular expressions).

7.c. Explain about BEGIN and END sections in awk, with an example program.

The *awk* statements are usually applied on all lines selected by address (selection criteria). But, if we want to print something before the processing starts or after completing the process, then BEGIN and END sections are useful.

The BEGIN and END sections are optional and have syntax as –

```
BEGIN{ action}
END { action}
```

The usage of these sections are depicted in the below given example. Here, in the BEGIN section we will print the heading for the columns and in the END section, we will print average basic pay. Let us first create a file *newPayroll.awk* (either using vi editor or cat command) as shown below –

newPayroll.awk

Note that, after the BEGIN section, we will write the selection criteria ($6>7500) and then the action within a pair of curly braces (flower brackets) as per the syntax of *awk* commands.

Now, run the file using the command –

```
$awk –F "|" –f newPayroll.awk emp.lst
BEGIN {
printf "SlNo \t Name \t\t Salary\n"
```

```
}
$6>7500{
count++
total += $6
printf "%3d %-20s %d\n", count, $2, $6
}
END{
printf "\nThe average salary is: %d\n", total/count
}
```

8.a. Explain the following commands.

i)export ii)eval iii)exec

export

The values stored in the shell variables are local to the shell and they are not passed on to the child shell. But, one can use ***export*** command to pass the variables used in the parent shell into the child shell.

```
x=10 #x is 10 on a parent shell
$ export x #parent x is exported to child
$ sh ex.sh #child shell is spawned to execute ex.sh
The value of x is 10 #value 10 got printed for x
The value of x is 20 #x is 20 after new assignment
$ echo $x
10 #value available with parent shell is printed
```

ii)eval

The ***eval*** command will take an argument and construct a command out of it, which will be executed by the shell. The ***eval*** statement tells the shell to take eval's arguments as command and run them through the command-line.

To understand the concept, let us first define few strings and then try to display the value of string with a numbered variable.

```
$ text1= "Emp ID:"
$ text1= "Name:"
$ text1= "Designation:"
$ x=1
$ echo $text$x
1 #output
```

Here, we expect the output to be Emp ID: because, $x is 1 and $text$x would be text1. But, the shell evaluates the command line from left to right. So, it first encounters $text which is not defined at all. Then it evaluates $x. Hence we will get the output as 1.

The ***eval*** statement evaluates a command line twice. In the first pass, it suppresses some evaluation and performs it only in the second pass. This is what we want in our previous example. So, if we escape the first $ symbol in $text$x, then the first pass evaluates only $x. So, we will get only text1. In the second pass, we have to evaluate it using ***eval*** as shown below –

```
$eval echo \$text$x
Emp ID: #displayed value of text1
```

Consider one more example in which we have a variable x assigned as 10 and another variable y which is assigned as x. Here, we would like to retrieve the value of x through y as shown below –

```
$ x=10
```

```
$ y=x
$ echo $y #it prints just x, but not 10
x
$ eval echo \$$y #escape first $ using \
10 #to get 10
```

In the above example, when we use the statement,

```
$ eval echo \$$y
```

the first $ symbol is escaped using slash. So, only $y is evaluated to get x. Then ***eval*** is used to evaluate value of x and result would be 10.

iii)exec

When ***exec*** command is used with another command externally on the command line, a new process is not spawned. Instead, the current process is overlaid with the new command. In other words, the exec command is executed in place of the current shell without creating a new process. This is useful for shell programmers when they need to overwrite the current shell itself with the code of another program. When the ***exec*** is preceded with any UNIX command, that command overwrites the current process – most of the times the shell. As the shell is overwritten by some other command, the user will be logged out immediately after the completion of that command.

Consider the following example –

```
$ exec date
Tue Jan 28 21:25:53 IST 2017
login:
```

Observe that, after displaying the date, login prompt is appeared. This indicates that the user is logged out because the *exec* has made the *date* command to overwrite the shell itself.

8.b. Desribe the in-built functions of awk.

There are several built-in functions in *awk* for performing arithmetic and string operations as shown in Table 4.2. Most of these functions behave similar to that in C programming. The arguments of the function are enclosed within parentheses and separated by comma. These functions are explained hereunder:

- **int(x):** This function calculates the integral portion of a number, without rounding it off. For example,

```
$awk 'BEGIN{ print int(3.7)}'
3
```

- **sqrt(x):** It is used to compute the square root of the number x.

```
$awk 'BEGIN{ print sqrt(25)}'
5
```

- **length:** It determines the length of its argument. If no argument is present, the entire line is assumed to be the argument. One can use it to locate the lines whose length exceeds a specific number of characters. The following example lists all the lines from the file *emp.lst* where number of characters is more than 50.

```
$awk -F"|" 'length>50' emp.lst
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
2365|barun sengupta|director|personnel|05/11/47|7800
4290|jayant Chodhury|executive|production|09/07/50|6000
6521|lalit chowdury|director|marketing|09/26/45|8200
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

Table 4.2 Built-in Functions in *awk*

Function	Description
int(x)	Returns integer value of x
sqrt (x)	Returns square root of x
length	Returns length of complete line
length(x)	Returns length of x
substr(stg, m, n)	Returns portion of string of length n, starting from position m in string *stg*
index(s1, s2)	Returns position of string s2 in string s1
split(stg, arr, ch)	Splits string *stg* into array *arr* using *ch* as delimiter and returns number of fields
system("cmd")	Runs UNIX command *cmd* and returns its exit status

 index(s1, s2): It determines the position of a string *s2* within a larger string *s1*. This function is usually helpful in validating single character fields. Consider the following examples:

Ex1. Checking for the character *b* in text *abcde*. The result displayed is 2.

```
$awk 'BEGIN{
> x=index("abcde", "b")
> print x}'
2
```

Ex2. Checking for substring *cd*, whose position is 3.

```
$ awk 'BEGIN{
> x=index("abcde", "cd")
> print x}'
3
```

substr(stg, m, n): This function extracts a substring of size *n* characters from a string *stg* starting from the position *m*. For example, following code extracts 3 characters from the string hello how are you? starting from 7th character.

```
$ awk 'BEGIN{
> x=substr("hello how are you?", 7,3)
> print x}'
how
```

This function can be used to select those people who have born between 1945 and 1951 using the following code –

```
$awk -F "|" 'substr($5, 7,2)>45 && substr($5,7,2)<52' emp.lst
9876|jai sharma|director|production|03/12/50|7000
2365|barun sengupta|director|personnel|05/11/47|7800
4290|jayant Chodhury|executive|production|09/07/50|6000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

 split(stg, arr, ch): This function breaks up a string *stg* on the delimiter *ch* and stores the fields in an array *arr[ ]*. The following example uses space (given within double quotes as a 3rd argument to *split* function) as a delimiter and stores each word of the string hello how are you? in the array arr.

```
$ awk 'BEGIN{
split("hello how are you?", arr, " ")
printf "%s\n%s\n%s\n%s\n",arr[1],arr[2],arr[3],arr[4]}'
```

hello
how
are
you?
 system: This function can be used for running UNIX commands within awk. For example,
$ awk 'BEGIN{
> system("date")
> system("pwd")}'
Mon Jan 1 08:54:14 IST 2007
/home/john

9.a. What are the privileges of the system administrator?

A system administrator (or super user or root user) has vast powers and access to almost everything in UNIX system. System admin is responsible for managing entire system like maintaining user accounts, security, disk space, backups etc.

Any user can acquire the status of superuser, if he/she knows the root password. For example the user *john* may become superuser using the command ***su*** as shown –

$ su
password: ******* #give root's password
pwd #working directory unchanged
/home/john

Following are some of the important privileges of a system administrator:

 Changing contents or attributes (like permission, ownership etc) of any file. He/she can delete any file even if the directory is write-protected.
 Initiate or kill any process.
 Change any user's password without knowing the existing password
 Set the system clock using ***date*** command.
 Communicate with all users at a time using ***wall*** command.
 Restrict the maximum size of files that users can create, using ***ulimit*** command.
 Control user's access to the scheduling services like ***at*** and ***cron***.
 Control user's access to various networking services like FTP, SSH (Secured Shell) etc.

9.b Explain the startup and shut down procedure in UNIX.

Whenever the system is about to start or about to shutdown, series of operations are carried out. We will discuss few of such operations here.

 Startup: When the machine is powered on the system looks for all peripherals and then goes through a series of steps that may take some time to complete the boot cycle. The first major event is the loading of the kernel (/kernel/genunix in Solaris and /boot/vmlinuz in Linux) into memory. The kernel then spawns ***init***, which in turn spawns further processes. Some of these processes monitor all the terminal lines, activate the network and printer. The ***init*** becomes the parent of all the shells.

A UNIX system boots to any one of two ***states*** (or mode) viz. single user mode or multiuser mode. This state is represented by a number or letter called as ***run level.*** The default run level and the actions to be taken for each run level are controlled by ***init***. The two states are discussed here –

o **Singer-user Mode:** The system admin uses this mode to perform administrative tasks like checking and backing up individual file systems.

Other users are prevented from working in single-user mode.
o **Multiuser Mode:** Individual file systems are mounted and system daemons are started in this mode. Printing is possible only in multiuser mode.
The ***who –r*** command displays the run level of our system:
$ who -r
. run-level 3 2007-01-06 11:10 last=S
The machine which runs at level 3 supports multiuser and network operations.
 Shutdown: While shutting down the system, the administrator has certain roles. The system shutdown performs following activities –
o Through the ***wall*** command notification is sent to all the users about system shutdown and asks to logout.
o Sends signals to all running processes to terminate normally.
o Logs off the users and kills remaining processes
o Un-mounts all secondary file systems.
o Writes file system status to disk to preserve the integrity of file system
o Notifies users to reboot or switch-off, or moves the system to single-user mode.

9.c. Explain about internal and external commands.
From the process point of view, the shell can recognize three types of commands as below–
 External Commands: The most commonly used UNIX utilities and programs like ***cat, ls, date*** etc. are called as external commands. The shell creates a process for each of such commands when they have to be executed. And, the shell remains as their parent.
 Shell Scripts: The shell spawns another shell to execute shell scripts. This child shell then executes the commands inside the script. The child shell becomes the parent of all the commands within that script.
 Internal Commands: Some of the built-in commands like ***cd, echo*** etc. are internal commands. They don't generate a process and are executed directly by the shell. Similarly variable assignment statements like x=10 does not create a child process.

10.a. Explain the following commands.
i)at ii)batch iii)nice iv)aliases
i)at
The ***at*** command takes one argument as the time specifying when the job has to be executed. For example,
$ at 10:44 #press enter by giving time
at> ls –l > out.txt #Give the command to be executed
at> <EOT> #Press [CTRL+d]
job 10 at 2018-01-01 10:44
The job goes to the queue and at 10.44 on Jan 1st, 2018 the command *ls –l* will be executed and the output would be stored in the file out.txt. One can check this file later for the output.
ii) batch
The ***batch*** command schedules the job when the system overhead reduces. This command does not use any arguments, but it uses internal algorithm to determine the

actual execution time. This will prevent too many CPU-hungry jobs from running at the same time.

$ batch < evalEx.sh #evalEx.sh will be executed later
job 15 at 2007-01-09 11:44

Any job scheduled with ***batch*** goes to special ***at*** queue from where it will be taken for execution.

iii)nice

In UNIX system, processes are executed with equal priority, by default. But, if high-priority jobs make demand, then they have to be completed first. The ***nice*** command with & operator is used to reduce the priority of jobs. Then more important jobs have greater access to the system resources.

To run a job with a low priority, use the command prefixed with ***nice*** as shown –

$nice wc emp.lst
15 31 741 emp.lst

iv)Aliases

The bash and ksh supports ***aliases*** (alternative names), which lets the user to assign any shorthand names to frequently used commands. This is done with the command ***alias***.

The ls –l command is used many times in UNIX. So, alias can be set for this command as –

$ alias el='ls –l'

Now onwards, in place of ls –l, user can use el. Note that, there must not be a space before and after = symbol in the above statement.

When we need to use ***cd*** command frequently to change directory to some long pathname as /home/john/ShellPgms/Scripts, then alias can be set as –

$ alias cdSys="cd /home/john/ShellPgms/Scripts"

So, we can just use cdSys to change the directory.

The alias set once can be revoked using the command ***unalias***. So, to unset the alias cdSys, use the statement as –

$ unalias cdSys

10.b. **Write an awk script to computer gross salary of an employee accordingly to rule given below.**

If basic salary is <10000 then HRA=15% of basic & DA=45% of basic if basic salary is>= 10000 then HRA=20% of basic & DA=50% of basic.

```
BEGIN{
        printf"enter the basic salary:\n"
        getline bs<"/dev/tty"
        if(bs<10000)
                {
                        hra=.15*bs
da=.45*bs
                }
                else
                {
                        hra=.2*bs
                        da=.5*bs
                }
        gs=bs+hra+da
        printf "gross salary=Rs.%.2f\n",gs
}
```