CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assessment Test 3 – Jan. 2020

Sub:	**Unix and Shell Programming**							**Sub Code:**	**18MCA12**
Date:	**13/1//2020**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**I**	**Branch:**	**MCA**

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
	PART I		CO	RBT
1	Explain the below mentioned commands with its usage and examples. i) tar ii) cpio iii) nice iv)kill v)at	[10]	CO5	L2
	OR			
2a)	Who is a super user? Explain the privileges of super user.	[5]	CO3	L1
b)	Explain briefly the commands used to manage the disk space in Unix	[5]	CO4	L2
	PART II			
3 a)	What is a process? Explain the mechanism of process creation.	[5]	CO2	L1
b)	Explain foreground and background processes in detail.	[5]	CO2	L2
	OR			
4	Write a note on i) batch ii) sh iii) export iv)eval v)exec	[10]	CO4	L2
	PART III			
5	Discuss the concept of conditional parameters substitution in shell programs	[10]	CO4	L2
	OR			
6	List the command that lets the user recall previously executed commands with example. Mention four different ways how this command can be used in Unix.	[10]	CO5	L2
	PART IV			
7a)	With a suitable input file, explain the general structure and the operational mechanism of an awk script.	[5]	CO4	L2
b)	Explain internal and external commands.	[5]	CO5	L2

	OR				
8	Write an awk script to delete duplicate line from text file. The order of original lines must remain unchanged.	[10]	CO4	L3
	PART V				
9	Why do we use history command? What is the output after each command i) grep http !30:4 iii)!find : s/pl/java ii) echo !$:r iv) !Cp :gs/doc/html	[10]	CO5	L3
	OR				
10a)	Explain UNIX start up and shut down process.	[5]	CO5	L2
b)	Demonstrate logical and relational operators in awk with suitable examples.	[5]	CO4	L2

1. Explain the below mentioned commands with its usage and examples.

i) tar   ii) cpio   iii) nice   iv)kill   v)at

i) tar

The ***tar*** is a tape archive command that was being used before the emergence of ***cpio***. It can create archives on tapes and floppy drives. Some features are –

 It doesn’t use standard input to obtain a file list
 It accepts file and directory names as arguments
 It copies one or more directory trees, in a recursive manner
 It can append to an existing archive, without overwriting.

Backing Up Files (-c)

The illustration of ***tar*** command for archiving files is –

```
# tar –cvf /dev/fd0 /home/john/*.sh
a /home/john/test.sh 1 tape blocks
a /home/john/caseEx.sh 1 tape blocks
a /home/john/evalEx.sh 5 tape blocks
```

The above command indicates that all the files with extension .sh in /home/john are archived into /dev/fd0. The character a at every line of output indicates the files are being appended into the device.

Restoring Files (-x)

Files archived in specific device are restored using –x option. When file/directory name is not specified, all the files from the backup device are restored.

```
# tar –xvf /dev/fd0
x /home/john/test.sh 1 tape blocks
x /home/john/caseEx.sh 1 tape blocks
x /home/john/evalEx.sh 5 tape blocks
```

Displaying the Archive (-t)

The –t option is used to display the contents of the device in a long format (similar to ls –l).

```
# tar –tvf /dev/fd0
rwxr-xr-x 203/50 472 Jun 4 10.23 2017 ./test.sh
rwxr-xr-x 203/50 583 Jun 4 11.51 2017 ./caseEx.sh
rwxr-xr-x 203/50 87 Jun 4 9.37 2017 ./evalEx.sh
```

Observe that file names are displayed as relative paths.

ii)cpio

The ***cpio*** (copy input-output) command copies files to and from a backup device. It uses standard input to take the list of file names. Then the files are copied along with their contents and header to standard output. Later, from standard output it is redirected to a file or a device. The ***cpio*** command uses two options viz. -o for output and –i for input, of which any one of them must be used.

5.15.1 Backing Up Files (-o)

The following example uses the ***ls*** command to generate list of all filenames as an input to ***cpio*** command. Then –o option is used to create archive on the standard output, which needs to be redirected to a device file.

```
# ls | cpio –ov > /dev/fd0
caseEx.sh
test.lst
```

```
emp.lst
here.sh
tputEx.sh
4 blocks #total size of the archive
```

The –v option displays each filename on the terminal while it is being copied. The above command compresses the files listed and stores the compressed file into /dev/fd0.

5.15.2 Restoring Files (-i)

Compressed files can be restored using –i option as shown below –

```
# cpio –iv < /dev/fd0
caseEx.sh
test.lst
emp.lst
here.sh
tputEx.sh
4 blocks
```

While restoring the subdirectories, the ***cpio*** command maintains the same subdirectory structures in the hard disk.

iii)nice

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) kill

The ***kill*** command sends a signal with the intention of killing one or more processes. It is an internal command in most of the shells. The external command **/bin/kill** is executed only when the shell lacks the kill capability. One or more PID are given as arguments to ***kill*** command and by default, it uses the SIGTERM (number 15) signal.

```
$kill 12815
```

v)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
```

2.a. Who is a super user? Explain the privileges of super user.

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.

2.b Explain briefly the commands used to manage the disk space in Unix

Reporting Free Space: df

The ***df*** command is used to check the amount of free space available in every file system.

```
$ df
Mounted on
Filesystem 1K-blocks Used Available Use% /dev/mapper/
16663980 6490020 9313800 42% /
/dev/hda3 101105 13684 82200 15% /boot
tmpfs 891796 0 891796 0% /dev/shm
none 891796 40 891756 1% /var/lib
```

This command displays file system, number of total blocks, used blocks, available space (in blocks and percentage) and in which directory, the file system is mounted.

5.14.2 Disk Usage: du

Usually, the user will be interested in the space consumption of a specific directory, rather than entire file system. The ***du*** command is used to check the usage of space in a specific directory recursively.

```
$ du /home/john
16 /home/john/.kde/Autostart
24 /home/john/.kde
16 /home/john/myDir
8 /home/john/.mozilla/extension
8 /home/john/.mozilla/plugins
24 /home/john/.mozilla
548 /home/john
```

This command lists the disk usage of each sub directory and finally gives the summary. If the list of sub-directories is very large and if we are interested only in the summary, ***-s*** option can be used as –

```
$ du -s /home/john
548 /home/john
```

If the system admin would likes know the space consumed by all the users, use the command as –

```
$du –s /home/*
```

3.a.What is a process? Explain the mechanism of process creation.

A process is an instance of a running program. A process is said to be born when the program starts execution and remains alive till the program is active.
There are three major steps in the creation of a process. Three important system calls or functions viz. ***fork, exec*** and ***wait*** are used for this purpose. The concept of process creation cycle will help to write the programs that create processes and helps in debugging shell scripts. The three steps are explained here –

- **Fork:** A process in UNIX is created with the help of system call ***fork.*** It creates a copy of the process that invokes it. The process image is identical to that of the calling process, except few parameters like PID. When a process is forked, the child gets a new PID. The forking mechanism is responsible for the multiplication of processes in the system.
- **Exec:** A new process created through *fork* is not capable of running a new program. The forked child needs to overwrite its own image with the code and data of the new program. This mechanism is known as ***exec***. The child process is said to exec a new program.
- **Wait:** The parent then executes the system call ***wait*** for the child process to complete. It picks up the *exit status* of the child and then continues with its other functions. A parent may not need to wait for the death of its child.

3.b Explain foreground and background processes in detail.

Every process has a parent process. If you run the command,

```
$ cat emp.lst
```

then the process representing ***cat*** command is created by the shell process. Now, the shell is said to be parent process of ***cat.*** The hierarchy of every process is ultimately traced to the first process (PID 0), which is set up when the system is booted. It is like the root directory of the file system.
A process can have only one parent, but multiple children. The following command creates two processes ***cat*** and ***grep*** both are spawned by the shell –

```
$cat emp.lst | grep 'director'
```

ps: PROCESS STATUS

The ***ps*** command is used to display some of the process attributes. This command reads the data structure of kernel and process tables to fetch the characteristics of processes. By default, ***ps*** display the processes owned by the user running the command. For example –

```
$ ps
PID TTY TIME CMD
11703 pts/1 00:00:00 bash
11804 pts/1 00:00:00 ps
```

After the head, every line shows the PID, terminal (TTY) with which the process is associated (controlling terminal), the cumulative processor time (TIME) that has been consumed since the process has been started and the process name (CMD).
UNIX is a multitasking system, which allows to run multiple jobs at a time. As there can be only one process in the foreground, the other jobs must run in the background. There are two different ways of running the jobs in background – using & operator and using ***nohup*** command. These methods are discussed here.

5.5.1 No Logging out: &

The & operator of shell is used to run a process in the background. The usage is as shown below –

```
$ sort –o newlist.lst emp.lst & #terminate line with & symbol
4080 # PID is displayed
$ #prompt is displayed to run another job
```

The & symbol at the end of the command line will make the job to run in the background. Immediately, the shell returns PID of the invoked command. And the parent shell will not wait till the job is completed (or the death of the child process), instead, the prompt is appeared so that a new job can be executed by the user. However, the shell remains as the parent of background process. Using & symbol, any number of jobs may be run in the background, if the system can sustain the load.

5.5.2 Log Out Safely : nohup

Normally, when the background processes are running, the user cannot logout. Because, shell is the parent of all the background processes and logging out kills the shell. And, by default when the parent shell is killed, all child processes will also die automatically. To avoid this, UNIX provides an alternative command ***nohup*** (no hangup). When this command is prefixed with any command, even after logging out, the background process keeps running. Note that the symbol & has to be used even in this case. For example,

```
$nohup sort emp.lst &
4154
nohup: appending output to ‘nohup.out’
```

Here, the shell returns the PID and some shells display the message as shown. When ***nohup*** command is run, it sends the standard output of the command to the file *nohup.out.* Now, one can safely logout the system, without aborting the command.

4. Write a note on

i) batch ii) sh iii) export iv)eval v)exec

i)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.

The *sh* Command

When a shell script is executed, the shell spawns (a child process created by parent process is known as spawning) a sub-shell and this sub-shell actually executes the script (Read Section 1.7.2 from the Module 1 for more explanation). When the script execution is completed the child shell is terminated and the control is returned to the parent shell.
One can explicitly invoke a sub-shell using ***sh*** command to execute a shell script. The syntax would be –

```
$sh filename.sh
```

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
```

5. Discuss the concept of conditional parameters substitution in shell programs

We know that to evaluate a variable in shell scripts, we will use $ symbol. We have also discussed earlier (Section 1.7.11 of Module1) that the ***expr*** command is used for evaluation of expression. But, Shell provides an interesting format to evaluate a variable depending on whether the variable has some defined value or a null value. This is known as ***parameter substitution***. The syntax is as given –

```
${<var>: <opt> <stg>}
```

Here, var is a variable to be evaluated
opt can be any one of the symbols +, -, = or ?
stg is a string

The behavior of the parameter substitution can be better understood based on the symbols with which it is used as discussed hereunder.

The + Option: If the var contains a defined value and not null, then it evaluates to stg. This can be understood in the following examples:

- **Ex1:** Assign a variable x with 10 and test for parameter substitution as shown –

```
$x=10
$echo ${x: + "x is defined"}
x is defined
```

As the value of x is not null, the string (stg) part in the expression got evaluated.

The – Option: This is quite opposite to + option. Here if the var do not contains a defined value and it is null, then it evaluates to stg. This is useful in setting default value for a variable/filename when user forgets to provide the value. But, the default value provided with – option is only for that moment, and var will not be assigned with that value. Consider following examples:

- **Ex1:** Use an un-assigned variable x and give it a default value.

```
$echo ${x: - 10}
10
$echo $x
#prints nothing
$
```

Here, the variable x was unassigned before. So, as it is null, value 10 will be assigned. But, in the prompt if we check its value now, it is still null only.

The = Option: This option goes one step ahead of – option, where it assigns the value to the variable. If we use = option inside the parameter substitution, then explicit assignment is not required.

- **Ex:**

```
$echo ${x:=10} #10 is assigned to x permanently
10
$echo $x #verify value of x
```

The ? Option: It evaluates the parameter if the variable is assigned and not null. Otherwise, it echoes a string and kills the shell. Consider the following script –

```
$ cat > test.sh
echo “Enter a file name:”
read fname
${fname: ? “No file name entered”}
echo fname
[Ctrl+C]
```

6. List the command that lets the user recall previously executed commands with example. Mention four different ways how this command can be used in Unix.

The ***history*** command in bash displays all commands that are previously executed, and in ksh, it displays last 16 commands. One can mention how many commands he/she would like to see, by giving the number as an argument to ***history*** command as –

```
$ history 5 #in bash
Or $ history -5 #in ksh
```

Accessing Previous Commands by Event Numbers (! And r) : The ! symbol (in ksh, we should use r) is used to repeat the previous command. To repeat the immediate previous command, use –

```
$ !! #it repeats previous command and executes.
```

The same job can be done with r in ksh.
If one needs to execute the specific command, the event number must be attached as –

```
$ !35 # executes command with event number 35
```

Executing Previous Commands by Context: It is quite difficult for any user to remember the event number of all the previous commands he/she used in past. But, there is a chance that he/she will be remembering first character/string of the command. The ! or r will help the user to recollect the commands by giving the specific character as below –

```
$ !g #executes last command which started with g (in bash)
Or $ r g # same task, in ksh
```

Substitution in Previous Commands: Frequently, user may need to execute any of the previous commands only after some modification in them – say changing the pattern in grep command, changing the string etc. This is possible with the help of modifier ***:s*** and the / as the delimiter between old and new patterns.
Consider an example – assume that, we had executed the *echo* command as –

```
$ echo $HOME $PATH
```

This would display the environment variables *HOME* and *PATH*. Now, we would like to replace PATH by SHELL. This can be done using the following command –

```
$ !echo:s/PATH/SHELL
echo $HOME $SHELL
/home/john /bin/bash
```

Here, the command has the meaning – “in the previous echo command, replace PATH by SHELL”

 Using Last Argument to Previous Command ($_): Frequently we use several commands on the same file. In such situations, instead of specifying filename every time, we can use $_ as the abbreviated filename. The $_ indicates that *last argument to previous command*. For example, if we create a directory *Test* using *mkdir* command and then we would like to change the directory to this new directory, we can use the command as –

```
$ mkdir Test # create new directory Test
$ cd $_ # change directory to Test
$ pwd # verify
/home/john/Test
```

7.a With a suitable input file, explain the general structure and the operational mechanism of an awk script.

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
```

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

```
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
}
```

7.b Explain 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.

Write an awk script to delete duplicate line from text file. The order of original lines must remain unchanged.

```
BEGIN {i=1}
{
flag=1;
for(j=1;j<i && flag ; j++)
{
if(x[j]==$0)
flag=0;
}
if (flag)
{
x[i]=$0;
printf("\n%s",x[i]);
}
i++;
}
```

8. Why do we use history command? What is the output after each command

iii) **grep http !30:4** **iii)!find : s/pl/java**

iv) **echo !$:r** **iv) !Cp :gs/doc/html**

The ***history*** command in bash displays all commands that are previously executed, and in ksh, it displays last 16 commands.

i) **grep http !30:4**
runs grep http with fourth argument of

ii) **!find : s/pl/java**
Repeats last find command after substituting java for pl

iii) **echo !$:r**
removes extension from previous commands filename

iv) **!Cp :gs/doc/html**
Removes last cp command after globally substituting html for doc

9. Explain UNIX start up and shut down process.

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.

10. Demonstrate logical and relational operators in awk with suitable examples.

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.
```

AWK supports the following logical operators −

Logical AND

It is represented by **&&**. Its syntax is as follows −

Syntax

```
expr1 && expr2
```

It evaluates to true if both expr1 and expr2 evaluate to true; otherwise it returns false. expr2 is
evaluated if and only if expr1 evaluates to true. For instance, the following example checks
whether the given single digit number is in octal format or not.

Example

```
[jerry]$ awk 'BEGIN {
   num = 5; if (num >= 0 && num <= 7) printf "%d is in octal format\n", num
}'
```

On executing this code, you get the following result −

Output

```
5 is in octal format
```

Logical OR
It is represented by ||. The syntax of Logical OR is −
Syntax
```
expr1 || expr2
```
It evaluates to true if either expr1 or expr2 evaluates to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to false. The following example demonstrates this −
Example
```
[jerry]$ awk 'BEGIN {
   ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n")
   print "Current character is whitespace."
}'
```
On executing this code, you get the following result −
Output
```
Current character is whitespace
```
Logical NOT
It is represented by **exclamation mark (!)**. The following example demonstrates this −
Example
```
! expr1
```
It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1. For instance, the following example checks whether a string is empty or not.
Example
```
[jerry]$ awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'
```
On executing this code, you get the following result −
Output
```
name is empty string.
```