CMR INSTITUTE OF TECHNOLOGY
DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS
INTERNAL ASSESSMENT-1 SOLUTION

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

i) cal ii) date iii) echo iv) who v) bc vi) printf

The cal command is a command line utility for displaying a calendar in the terminal. It can be used to print a single month, many months or an entire year. It supports starting the week on a Monday or a Sunday, showing Julian dates and showing calendars for arbitrary dates passed as arguments.

```
cal
   September 2016
Mo Tu We Th Fr Sa Su
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

The date command displays the current date and time. It can also be used to display or calculate a date in a format you specify.**date** with no options will output the system date and time, as in the following output:
Thu Feb 8 16:47:32 MST 2001

to display a line of text/string on standard output or a file.To print string "Hello, World!" on console

```
$ echo "Hello, World!"
```

output:
Hello, World!

Who
The **who** command prints information about all users who are currently logged in.

Bc
bc is an arbitrary-precision language for performing math calculations.
Printf
printf prints a *formatted string* to the standard output. Its roots are in the C programming language, which uses a function by the same name. It is a handy way to produce precisely-formatted output from numerical or textual arguments.

```
printf "Hi, I'm %s.\n" $LOGNAME
```

2. A. Explain the following looping statements in shell script.

i) if ii)while

The **if...else...fi** statement is the next form of control statement that allows Shell

to execute statements in a controlled way and make the right choice.

Syntax

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

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

```
#!/bin/sh


a=10

b=20

if [ $a == $b ]then

   echo "a is equal to b"else

   echo "a is not equal to b"fi
```

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

```
#!/bin/sh


a=0

while [ $a -lt 10 ]do

   echo $a

   a=`expr $a + 1`done
```

B. Explain Shell variables : i) $HOME ii) PWD

The name of a user's home directory is by default identical to that of the user. Thus, for example, a user with a user name of *mary* would typically have a home

directory named *mary*. It would have an *absolute pathname* of */home/mary*. An absolute pathname is the location of a directory or file relative to the root directory, and it always starts with the root directory (i.e., with a forward slash).
$HOME

- **pwd** --- tells you where you currently are.

pwd is a shell builtin

Print the name of the working directory. If any of the subdirectories in the path are symbolic links, and you used the symlink names when changing to the directory, the symlink names are printed. Example output:
/home/hope/actual_directory_name/actual_subdirectory/mydir

The **Unix script command**. **script** is used to take a copy of everything which is output to the terminal and place it in a log file. It should be followed by the name of the file to place the log in, and the exit **command** should be used to stop logging and close the file.

3. Explain the below mentioned commands with its usage and examples.
i) spell ii) script iii) sleep iv) uname v) passwd

spell is a very minimalistic spell-checking program, based on the original UNIX spell checker. It reads the contents of file *FILE*, word for word, checking them against its dictionary. If a word does not correspond with any of **spell**'s dictionary words, the word is printed.
If an input *FILE* is not specified, or is specified as a dash ("**-**"), **spell** performs a spell check of standard input.

sleep is a **command in Unix**, **Unix**-like and other operating systems that suspends program execution for a specified time. The **sleep**instruction suspends the calling process for at least the specified number of seconds (the default), minutes, hours or days.
Uname is a computer program in **Unix** and **Unix**-like computer operating systems that prints the name, version and other details about the current machine and the operating system running on it. The **uname** system call and **command** appeared for the first time in PWB/**UNIX**. Both are specified by POSIX.
The **passwd command** is used to change the**password** of a user account. A normal user can run**passwd** to change their own **password**, and a system administrator (the superuser) can use**passwd** to change another user's **password**, or define how that account's **password** can be used or changed.
4. A)Briefly explain for loop in shell script with examples.
The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9do

    echo $vardone
```

to display all the files starting with **.bash** and available in your home. We will execute this script from my root −

```
#!/bin/sh

for FILE in $HOME/.bash*do

    echo $FILEdone
```

B) Explain case conditions with an example.

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
    pattern1)
        Statement(s) to be executed if pattern1 matches
        ;;
    pattern2)
        Statement(s) to be executed if pattern2 matches
        ;;
    pattern3)
        Statement(s) to be executed if pattern3 matches
        ;;
    *)
       Default condition to be executed
       ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

```
#!/bin/sh



FRUIT="kiwi"

case "$FRUIT" in

    "apple") echo "Apple pie is quite tasty."

    ;;
```

```
"banana") echo "I like banana nut bread."

;;

"kiwi") echo "New Zealand is famous for kiwi."

;;esac
```

5. A)Briefly explain how to create and delete directory with examples.

Creating directories

To create a new directory, use the mkdir command. The follow example creates a new directory named 'directory_name':

```
[server]$ mkdir directory_name
```

Deleting directories

There are actually a few ways to delete directories in the shell. To delete an empty directory, use the rmdir command:

```
[server]$ rmdir directory_name
```

C) With suitable example bring out the difference between absolute and relative pathnames

While file names are certainly important, there is another important related concept, and that is the concept of a **file specification**[1] (or file spec for short). A file spec may simply consist of a file name, or it might also include more information about a file, such as where is resides in the overall file system. There are 2 techniques for describing file specifications, **absolute** and **relative**.

With **absolute** file specifications, the file specification always begins from the root directory, complete and unambiguous. Absolute file specs are sometimes referred to as fully qualified path names[2]. Thus, absolute file specs always begin with /. For example, the following are all absolute file specs from the diagram above:

/etc/passwd
/bin
/usr/bin
/home/mthomas/bin
/home/mthomas/class_stuff/foo

Note the the first slash indictes the top of the tree (root), but each succeeding slash in the file spec acts merely as a separator. Also note the files named bin in the file specifications of /bin, /usr/bin, and /home/mthomas/bin are different bin files, due to the differing locations in the file system hierarchy.

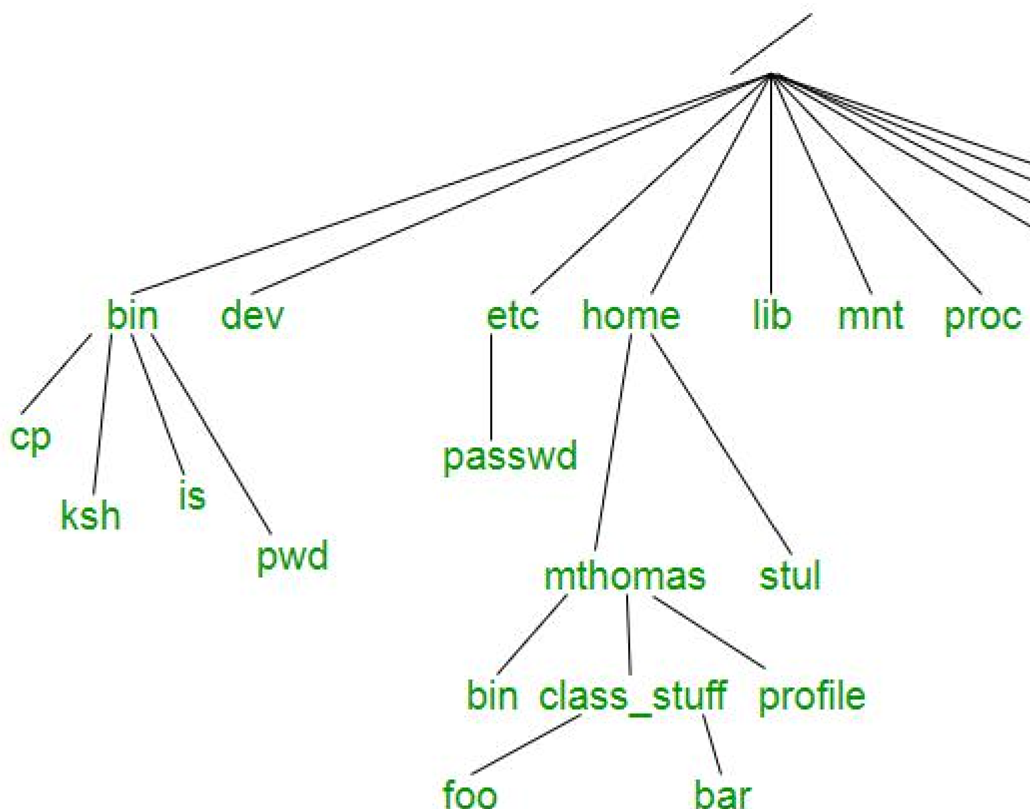
With **relative** file specifications, the file specification always is related to the users current position or location in the file system. Thus, the beginning

(left-most part) of a relative file spec describes either:

- an ordinary file, which implies the file is contained within the current directory
- a directory, which implies a child of the current directory (i.e. one level down)
- a reference to the parent of the current directory (i.e. one level up)

6. A)Briefly explain the unix file system.
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 filesystem, 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.
- **/var/tmp :** A place for temporary files which should be preserved between system reboots.

B)What is a file? Explain the categories of files found in UNIX Operating System.
From a user perspective in a Unix system, everything is treated as a file. Even such devices such as printers and disk drives.

How can this be, you ask? Since all data is essentially a stream of bytes, each device can be viewed logically as a file.

All files in the Unix file system can be loosely categorized into 3 types, specifically:

1. ordinary files
2. directory files
3. device files [1]

While the latter two may not intuitively seem like files, they are considered "special" files.

The first type of file listed above is an ordinary file, that is, a file with no "special-ness". Ordinary files are comprised of streams of data (bytes) stored on some physical device. Examples of ordinary files include simple text files, application data files, files containing high-level source code, executable text files, and binary image files. Note that unlike some other OS implementations, files do not have to be binary Images to be executable (more on this to come).

The second type of file listed above is a special file called a directory (please don't call it a folder?). Directory files act as a container for other files, of any category. Thus we can have a directory file contained within a directory file (this is commonly referred to as a subdirectory). Directory files don't contain data in the user sense of data, they merely contain references to the files contained within them.

It is perhaps noteworthy at this point to mention that any "file" that has files directly below (contained within) it in the hierarchy **must** be a directory, and any "file" that does not have files below it in the hierarchy can be an ordinary file, or a directory, albeit empty.

The third category of file mentioned above is a device file. This is another special file that is used to describe a physical device, such as a printer or a portable drive. This file contains no data whatsoever, it merely maps any data coming its way to the physical device it describes.

[1] Device file types typically include: character device files, block device files, Unix domain sockets, named pipes and symbolic links. However, not all of these file types may be present across various Unix implementations.

7.A)Write significance of the following commands
i) trap ‘rm $$* ; echo “program interrupted” ; exit’ 1 2 15
When the trap command is issued, it removes all the temporary files and prints a message “echo interrrupted” and exit with any of the interrupt signals 1 2 and 15
ii) date | cut –d” “ –f1

Date command gives the output as Thu Feb 8 16:47:32 MST 2001.The pipe symbol is used to supply the output of right side to the input of left side. And so the output of date command is given as input to cut command. The parameters of cut command are -d as demiliter and the delimiter used is space and it outputs the first field which is Thu.

B)Explain the parent-child relationship.

To identify where we are, we type and the system returns the following:

$ pwd *[Enter]*

/home/mthomas/class_stuff

Thus the parent of this directory is:

/home/mthomas # in absolute form
.. # in relative form

Looking at another example:

$ pwd *[Enter]*
/home/mthomas

Thus the parent of this directory is:

/home # in absolute form
.. # in relative form

And one (note there could be many) child of the /home/mthomas directory is:

/home/mthomas/bin # in absolute form
bin # in relative form

So you ask "How the heck do we use this?" One uses this to navigate or move about the file system. Moving about the file system is accomplished by using the **cd** command, which allows a user to **c**hange **d**irectories. In the simplest usage of this command, entering

$ cd *[Enter]*

will move the user to their "home" or login directory (as specified by the $HOME variable [4]). If a user wishes to change to another directory, the user enters

$ cd file_spec *[Enter]*

and assuming file_spec is a valid directory, the users current working directory will now be this directory location. Remember, the file specification can always be a relative or an absolute specification.

As before, we type and the system returns the following:

$ pwd *[Enter]*
/home/mthomas/class_stuff

If we wish to change directories to the /home/mthomas/bin directory, we can type

$ cd /home/mthomas/bin *[Enter]* # absolute, will work from anywhere

or

$ cd .. *[Enter]* # relative, move up one directory, i.e. to the parent
$ cd bin *[Enter]* # relative, move down to the bin directory

or

$ cd ../bin *[Enter]* # relative, both steps in one file spec

7. A)How do you set terminal characteristics? Explain with examples.

stty command is used to manipulate the terminal settings. You can view and modify the terminal settings using this command as explained below. Display All Settings

-a option displays all the stty settings in a user friendly readable format as shown below.

```
# stty -a

speed 38400 baud; rows 59; columns 208; line = 0;

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ; eol2 = ; swtch = ; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;

-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts -cdtrdsr

-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -ixany -imaxbel -iutf8

opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0

isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt ech
```

B) Explain exit status of a command with examples.
Following the execution of a pipe, a $? gives the **exit status** of the **last command**executed. After a script terminates, a $? from the **command**-line gives the **exit status** of the script, that is, the **last command** executed in the script, which is, by convention, 0 on success or an integer in the range 1 - 255 on error.

```
#!/bin/bash

echo hello
echo $?      # Exit status 0 returned because command executed successfully.

lskdf        # Unrecognized command.
echo $?      # Non-zero exit status returned -- command failed to execute.

echo

exit 113     # Will return 113 to shell.
```

C)Briefly explain positional parameters with examples

A positional parameter is a variable within a shell program; its**value** is set from an argument specified on the **command** line that invokes the program. Positional parameters are numbered and are referred to with a preceding ``$'': $1, $2, $3, and so on

```
$ cat pp
  echo   The first positional parameter is: $1
  echo   The second positional parameter is: $2
  echo   The third positional parameter is: $3
```

echo The fourth positional parameter is: $4
$

```
#!/usr/bin/ksh


echo "The total no of args are: $#"

echo "The script name is : $0"

echo "The first argument is : $1"

echo "The second argument is: $2"

echo "The total argument list is: $*"

[root@gpr ~]# ./cmd 1 2 3 4

The total no of args are: 4

The script name is : ./cmd

The first argument is : 1

The second argument is: 2

The total argument list is: 1 2 3 4

[root@gpr ~]#
```