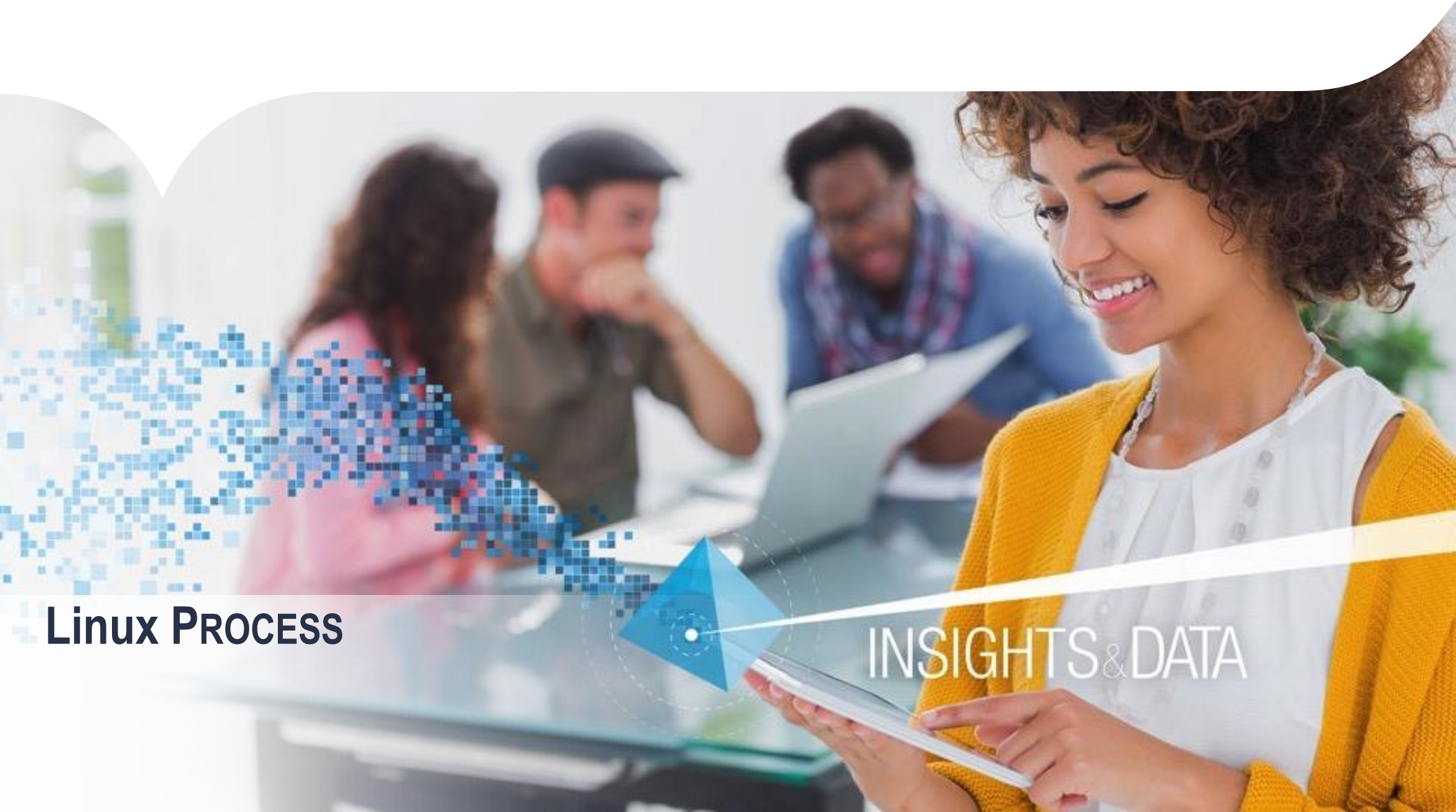
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

People matter, results count.



Linux PROCESS

UNIX/LINUX

PROCESS

OBJECTIVES

In this session, you will learn to:

- Use process-related commands like
  ps, **kill**, **sleep**
- Start a background process
- Use background and foreground-related commands like
  bg, **fg**, **jobs , nice , nohup**

PROCESSES

- **Process** - a program in execution\
- When program is executed, a new process is created
- The process is alive till the execution of the program is complete
- Each process is identified by a number called **pid**

LOGIN SHELL

As soon as the user logs in, a process is created which executes the login shell.

Login shell is set for each login in **/etc/passwd** file.

- Eg. $ echo $$

  11014

PS

- The **ps** command is used to display the characteristics of a process

- It fetches the **pid**, **tty**, **time,** and the command which has started the process.
  - **-f** lists the pid of the parent process also.
  - **-u** lists the processes of a given user
  - **-a** lists the processes of all the users
  - **-e** lists all the processes including the system processes

BACKGROUND PROCESS

- Enables the user to do more than one task at a time.
- If the command terminates with an ampersand (&), UNIX executes the command in the background
- Shell returns by displaying the process ID (PID) and job id of the process
- Eg: $ sort -o emp.lst emp.dat& - it runs background process

  [12] 13716 - this is process ID

Background Process can be seed by entering $ps –f

BACKGROUND PROCESS

- **nohup**
  - Lets processes to continue to run even after logout
  - The output of the command is sent to *nohup.out* if not redirected

$ nohup *command args*

```
$ nohup sort emp.lst &
[1] 21356
nohup: appending output to `nohup.out'
```

BACKGROUND PROCESS

- $ nohup sort emp.dat& - runs background process even after log out and output stores in

  nuhup.out file

  [12] 28656

- $ nohup: appending output to `nohup.out'

  [12] Done nohup sort emp.dat

- $ cat nohup.out – Out put can be seen from nohup.out

BACKGROUND PROCESS (CONT…)

- $ nohup sort emp.dat > mynohup.out&

  [12] 23050

- [12] Done nohup sort emp.dat >mynohup.out

- $ cat mynohup.out

- It holds output into user defined output file

BACKGROUND PROCESS

- **wait command**
  - can be used when a process has to wait for the output of a background process
  - The **wait** command, can be used to let the shell wait for all background processes terminate.

    $ wait

  - It is possible to wait for completion of one specific process as well.

CONTROLLING BACKGROUND PROCESSES

- **jobs**
  - List the background process
- **fg % <job id>**
  - Runs a process in the foreground
- **bg %<job id>**
  - Runs a process in the background
  - Eg: $ sleep 3000& - runs background process (assuming jobid – 15)

    $ jobs - view all job's status

    $ fg %15 - bring /runs job 15 from background to forground

    $ ^z – to stop forground process job

    $ bg %15 – again keeps job15 to background

PROCESS PRIORITY

- **nice**
  - Used to reduce the priority of jobs

- **To run a job with a low priority, the command name should be prefixed with nice**

  Eg: $ nice sleep 3000&

  [16] 22552

THE KILL COMMAND

- kill: Kills or terminates a process
- kill command send a signal to the process
  - The default signal is 15
- kill -9 <Pr. Id> it is sure kill
  - Terminates the process abruptly
- Killing more than one process
  - $ kill <Pr.id1> <Pr.id2>……
  - Eg: $kill 109 122 145

QUIZ

Which command is used to terminate a process?

A. shutdown

B. haltsys

C. cancel

D. kill

Option D

Which of the following commands is used to view your file 24 lines at a time?

A. pg
B. cat
C. lp
D. /p

Option A

QUIZ

The command which continues a stopped job by running it in the foreground.

A. bg

B. fg

C. wait

D. sleep

Option B

Lists the jobs that you are running in the background and in the foreground

A. bg

B. fg

C. list

D. jobs

Option D

Q & A

SUMMARY

- In this session, you learned to:
  - Define a process
  - Use process-related commands like ps, kill, sleep
  - Start a background process
  - Use background and foreground-related commands like bg, fg, jobs

UNIX/LINUX

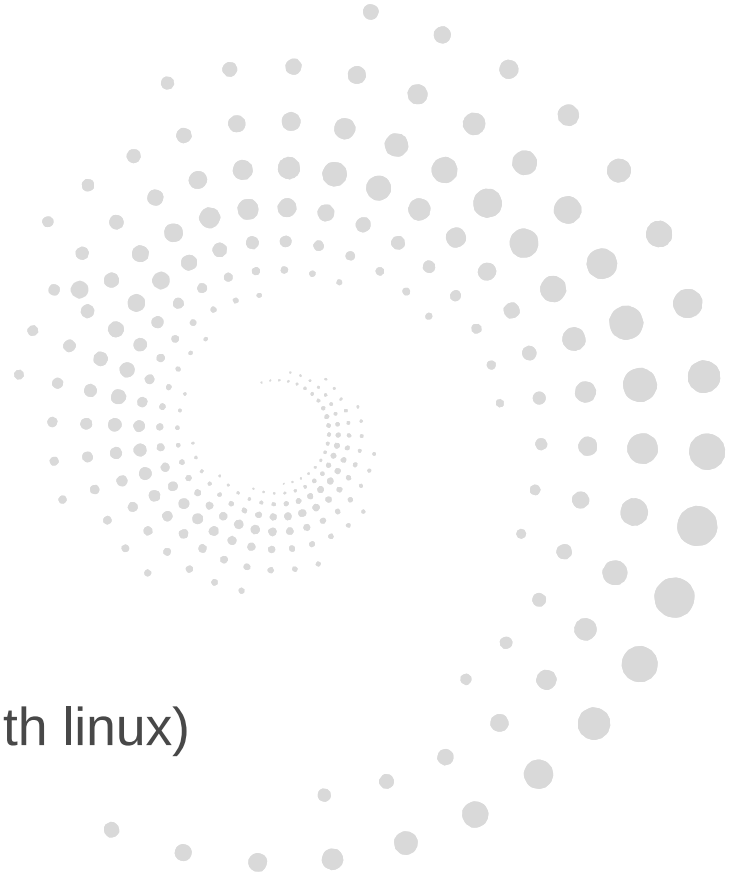
UNIX SHELL PROGRAMMING

OBJECTIVES

- In this session, you will learn to:
  - Use Shell variables
  - Write scripts to process positional parameters
  - Use “test” command
  - Use “if” construct
  - Use “for” loop
  - Use “while” loop
  - Use “case” construct
  - Define and use functions
  - Debug shell scripts

FLAVOURS OF THE UNIX SHELL

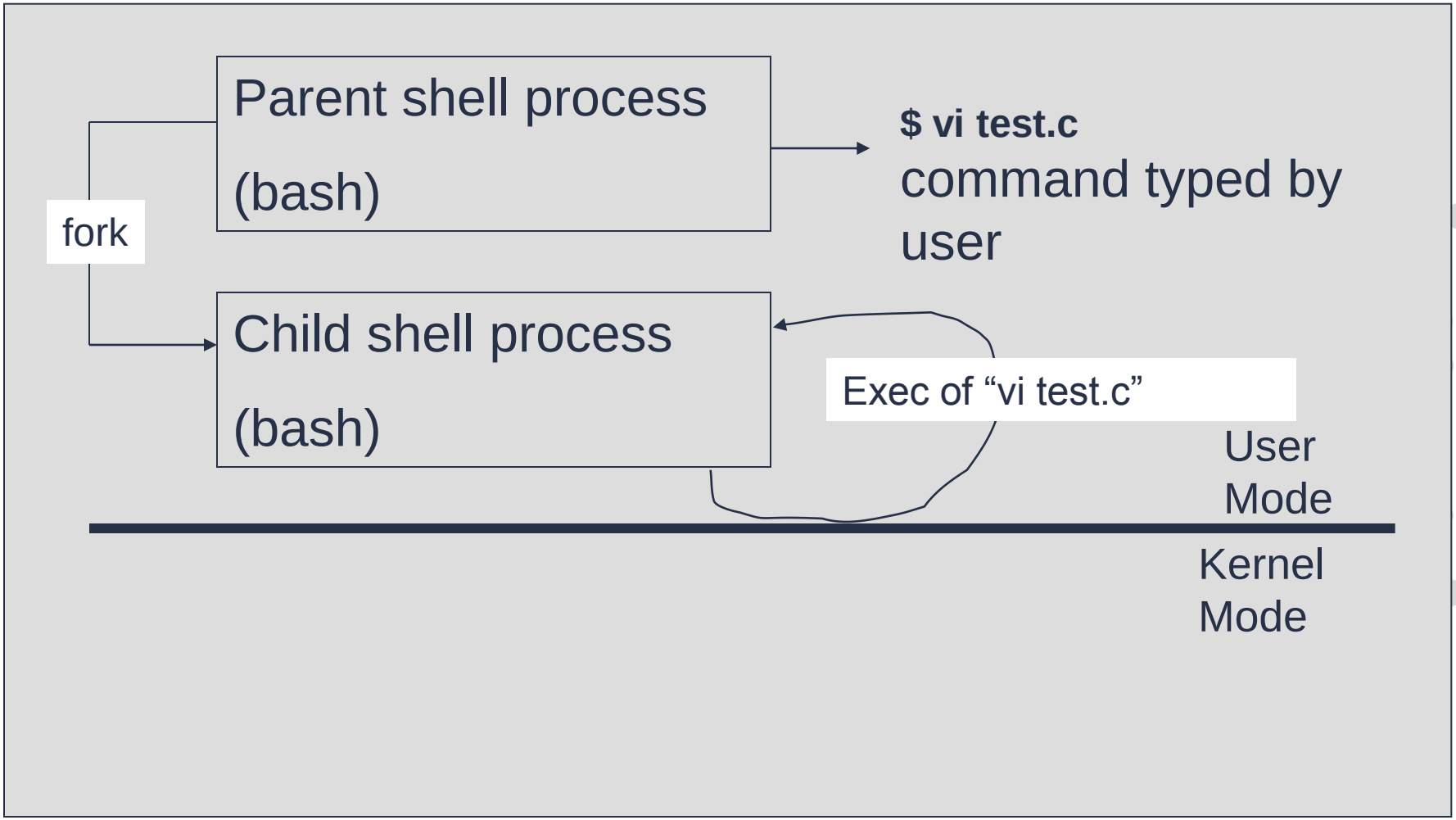
- Bourne shell sh
- C shell csh
- Korn shell ksh
- Bourne again shell bash (shell distributed with linux)



COMMAND PROCESSING

- Displays the shell prompt and reads the command typed by the user.
- Interprets the command and classifies it as an internal (built-in), or an external command.
- If it is NOT a built-in command, searches for the command in the **PATH-specified** directories, and executes that command if it is found.

SHELL FEATURES



Additional Shell Features

- **Each shell, apart from the basic features, provides additional features such as:**
  - Maintaining command history (C, korn and bash)
  - Renaming (aliasing) a command (C, korn, bash)
  - Command editing (C, korn and bash)
  - Programming language (all shells)

HISTORY

- some UNIX shells support command history
- facility to keeps track of commands that were executed
- facility to rerun previously executed commands
- bash shell supports the following

!!	recall the last command and execute it.
!num	execute n^{th} command where n is the **num** specified after **!**

ALIAS

- alias can be used to give new name to an existing command
- A better name that represents a single command or a sequence of commands to be executed, often with appropriate options
- **alias** is an internal command

alias newname=command

$ alias l='ls –l'

The unalias command cancels previously defined alias.

FILE NAME SUBSTITUTION

- When the user enters a command string, the shell parses the string into following components:
  - Command (the first part of the string, till the first space char)
  - Command arguments (the subsequent parts of the string)

- For example, given the command-string “ls –l *.c”, this string contains the “ls” command and two arguments “-l” and “*.c”.

FILE NAME SUBSTITUTION

- In arguments of a command, the shell recognizes certain characters – such as *****, **?**, **[ ]**, and **-** as special characters and expands these characters into a filename list before executing the command.

- To see how this works, enter following commands while in /bin directory

$ ls a*

$ ls ??

SHELL PROGRAMMING

- Allows
  - Defining and referencing variables
  - Logic control structures such as if, for, while, case
  - Input and output

SHELL VARIABLES

- A variable is a name associated with a data value, and it offers a symbolic way to represent and manipulate data variables in the shell. They are classified as follows
  - user-defined variables
  - environment variables
  - predefined variables

- value assigned to the variable can then be referred to by preceding the variable name with a **$** sign.

SHELL VARIABLES

- The shell provides the facility to define normal, and environment variables.
- A normal variable can be only used in the shell where it is defined.
- An environment variable can be used in the shell where it is defined, plus any child shells invoked from that shell.

USING NORMAL VARIABLES

To define a normal variable, use the following syntax:

- variable_name=value

Examples:

x=10

textline_1=‘This line was entered by $USER’

textline_2=“This line was entered by $USER”

allusers=`who`

usercount=`who | wc –l`

USING NORMAL VARIABLES

- Once variables are defined, one can use the echo command to display the value of each variable:

```
echo $x
echo $textline_1
echo $textline_2
echo $allusers
echo $usercount
```

USING ENVIRONMENT VARIABLES

- To define an environment variable, use following syntax:

  variable_name=value

  export variable_name

- Examples:

  $ x=10; export x

  $ allusers=`who` ; export allusers

BUILT-IN ENVIRONMENT VARIABLES

PATH

BASH_ENV

HOME

PWD

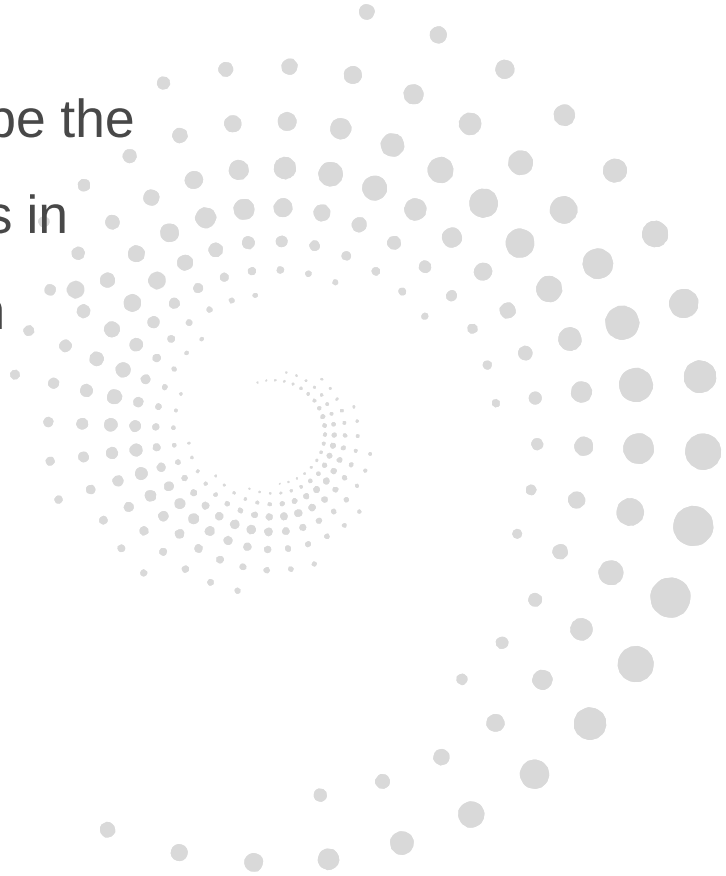
SHELL

TERM

- MAIL
- USER
- LOGNAME
- PS1
- PS2

SAMPLE SHELL SCRIPT

```
#! /bin/bash
# The above line has a special meaning. It must be the
# first line of the script. It says that the commands in
# this shell script should be executed by the bash
# shell (/bin/bash).
# ------------------------------------------------------
echo “Hello $USER….”
echo “Welcome to programming shell scripts..”
# ------------------------------------------------------
```



EXECUTING SHELL SCRIPTS

There are two ways of executing a shell script:

By passing the shell script name as an argument to the shell. For example:

sh script1.sh

If the shell script is assigned execute permission, it can be executed using it's name. For example:

./script1.sh

PASSING PARAMETERS TO SCRIPTS

- parameter can be passed to a shell script
- parameters are specified after the name of the shell script when invoking the script.
- Within the shell script, parameters are referenced using the predefined variables **$1** through **$9**.
- In case of more than 9 parameters, other parameters can be accessed by **shift**ing.

BUILT-IN VARIABLES

- Following are built-in variables supported
  - $0, $1…$9 - positional arguments
  - $* - all arguments
  - $@ - all arguments
  - $? - exit status of previous command executed
  - $$ - PID of the current process
  - $! - PID of the last background process

PASSING PARAMETERS TO SCRIPTS

- Consider following shell script:

```
----------------------script2.sh-------------------------
echo “Total parameters entered: $#”
echo “First parameter is : $1”
echo “The parameters are: $*”
shift
echo “First parameter is : $1”
-----------------------------------------------------------
```


- Execute the above script using the **“script2.sh these are the parameters”** command.

PASSING PARAMETERS TO SCRIPTS

- The shell parameters are passed as strings.
- to pass a string containing multiple words as a single parameter, it must be enclosed within quotes.
- For example,

```
$ ./script2.sh “this string is a single parameter”
```

DOING ARITHMETIC OPERATIONS

- Arithmetic operations within a shell script can be performed using **expr** command.

- Example,

```
x=10
y=5
number_1 = `expr $x + $y`
number_2 = `expr $x - $y`
number_3 = `expr $x / $y`
number_4 = `expr $x \* $y`
number_5 = `expr $x % $y`
```

USING THE TEST COMMAND

The general syntax of **test** command is:

test *expression*

- The *expression* can be formed using a combination of shell variables and the operators supported by the test command. These operators provide facility to compare numbers, string and logical values, file types and file access modes.

USING THE TEST COMMAND

To compare two integers using **test** following operators are available:

-eq (equal to)

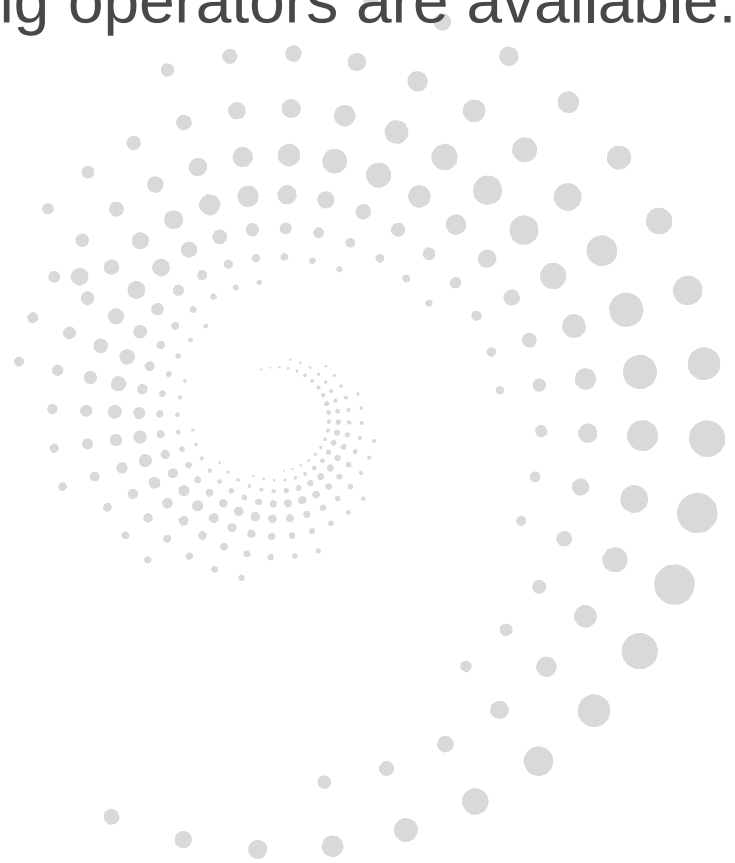
-ne (not equal to)

-lt (less than)

-le (less than or equal to)

-gt (greater than)

-ge (greater than or equal to)



USING THE TEST COMMAND

- General syntax

test *expression*

or

[expression]

test integer1 operator integer2

OR

[integer1 operator integer2]

USING THE TEST COMMAND

To compare two strings using the **test** command, following operators are available:

- **string1 = string2** (equal to, please note it is a single **=**)
- **string1 !**= **string2** (not equal to)
- **string1** (string is not NULL)
- **-n string1** (string is not NULL and exists)
- **-z string1** (string is NULL and exists)

USING THE TEST COMMAND

The syntax for this string comparison is:

test string1 operator string2

OR

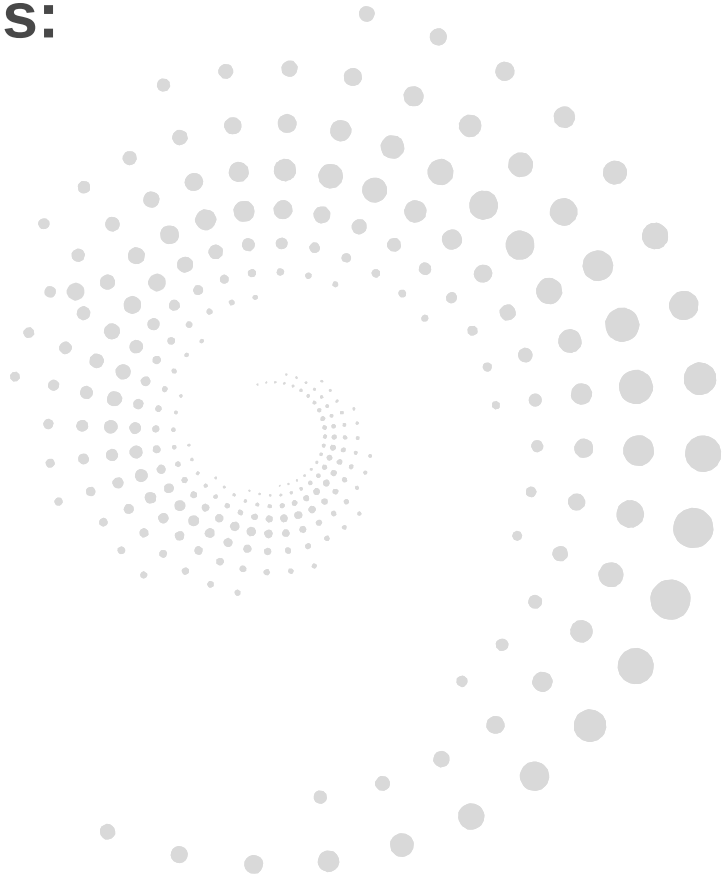
[string1 operator string2]

OR

test operator string

OR

[operator string]



USING THE TEST COMMAND

To check a file type/access permissions using the **test** command, following operators are available:

- **-s file** (file is not empty and exists)
- **-f file** (Ordinary file and exists)
- **-d file** (file is a directory and exists)
- **-r file** (file is readable and exists)
- **-w file** (file is write-able and exists)
- **-x file** (file is executable and exists)

USING THE TEST COMMAND

To check a file type/access permissions using the **test** command, following operators are available:

- **-b file** (file is a block device and exists)
- **-c file** (file is a character device and exists)
- **-p file** (file is a named pipe and exists)
- **-g file** (file has sticky bit set)
- **-u file** (file has **setuid** bit set)
- **-t file_des** (file descriptor is standard output)

COMBINING CONDITIONS

- It is possible to combine conditions by using following operators:
  - **-a** (logical AND operator)
  - **-o** (logical OR operator)
  - **!** (logical NOT operator)

COMBINING CONDITIONS

The syntax for this is:

test expression_1 –a expression _2,

OR

[expression _1 –a expression _2]

test expression_1 –o expression _2,

OR

[expression_1 –o expression_2]

test ! expression _1

OR

[! expression _1]

CONDITION CHECKING IN SCRIPTS

Bash shell provides the **if** command to test if a condition is true. The general format of this command is:

```
if condition
then
   command
fi
```

The **condition** is typically formed using the **test** command.

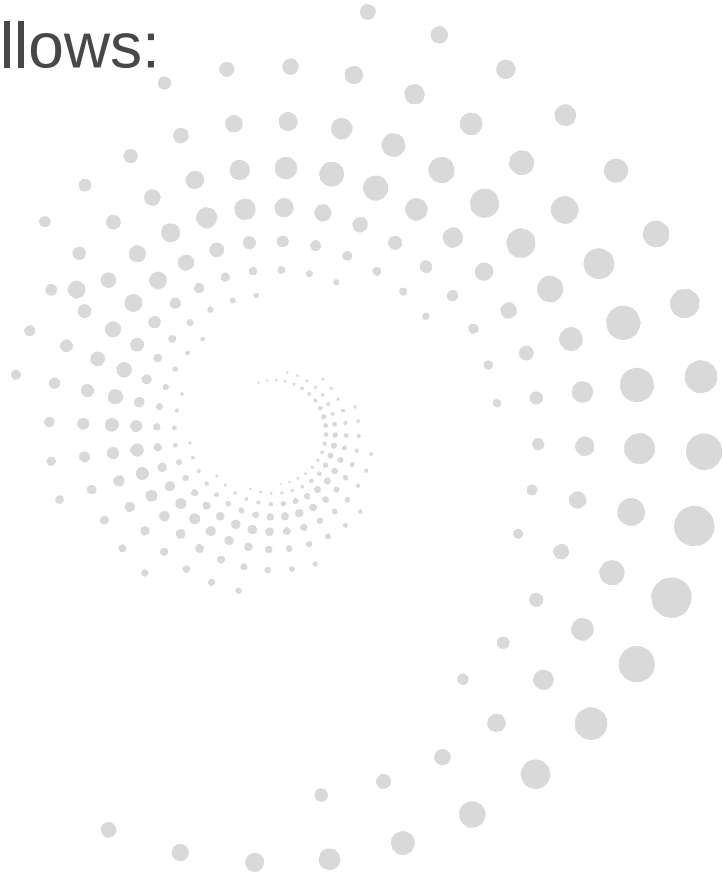
EXAMPLE

```
# to check if the current directory is the same as your home directory
curdir=`pwd`
if test “$curdir” !=  “$HOME”
then
   echo your home dir is not the same as your present working directory
else
echo $HOME is your current directory
fi
```

CHECKING MULTIPLE CONDITIONS

The complex form of **if** statement is as follows:

```
if condition_1
then
    command
elif condition_2
then
    command
else
    command
fi
```



QUIZ

1.The operator used to check whether two string are eqal or not

A. string1 = string2

B. string1 != string2

C. String1

D. none of the above

Option A

2.Which command is used to compare two strings/integers

A. Test

B. Compare

C. Equal

D. Not equal

Option A

QUIZ

The option used to check whether a file is block device or not

A. -b

B. -r

C. -c

D. -p

Option A

The option used to check whether a file has sticky bit or not

A. -u

B. -g

C. -t

D. -b

Option B

USING FOR LOOP

The Bash shell provides a **for** loop.

The **syntax** of this loop is:

```
for variable in list
do
        command
            ...
        command
done
```

Example:

```
for i in 1 2 3 4 5
do
   echo -n $i \* $i = "  "
   echo `expr $i  \*  $i `
done
```

EXAMPLE

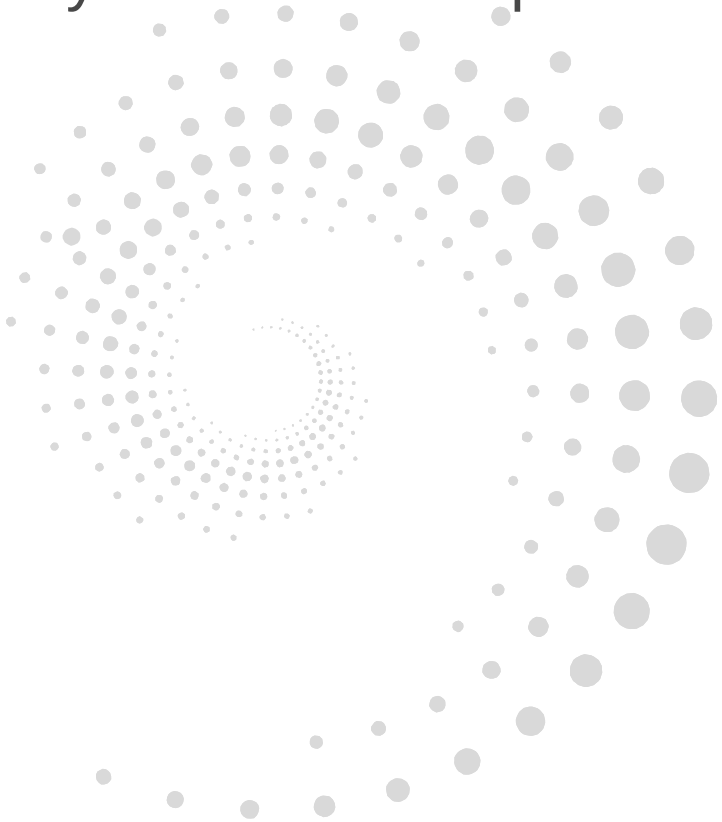
----------script.sh----------

```
#! /bin/sh
usernames=`who | cut –d “ “ –f1`
echo “Total users logged in = $#usernames”
#
for user in ${usernames}
do
        echo $user
done
```

USING WHILE LOOP

The Bash shell provides a **while** loop. The syntax of this loop is:

```
while condition
do
        command
            ...
        command
done
```



Example

Shell script checks for a blank/non blank string

```
eg:     read nam
        while [ -z “$nam” ]
        do
            read nam
        done
        echo the string is $nam
```

the above piece of code keeps accepting string variable **nam** until it is non zero in length.

EXAMPLE

Shell script to compute factorial of a given number

```
#!/bin/bash
n=$1
if [ $n -eq 0 ]; then
  fact=0
else
  fact=1
while [ $n –ne 0 ]
do
     fact=`expr $fact \* $n`
     n=`expr $n – 1`
  done
fi
echo $fact
```

The case Statement

The structure of case statement

```
case    value in
pattern1)
        command
        command;;
pattern2)
        command
        command;;
patternn)
        command;;
esac
```

EXAMPLE

```
#!/bin/bash

echo enter 2 nos

read num1

read num2

echo “enter 1 (for addition) or 2 (for
   subtraction)

read choice
```

(contd.)

```
case $choice in
1)  res=`expr $num1 + $num2`
    echo result is $res;;
1)  res=`expr $num1 -  $num2`
    echo result is $res;;
*) echo invalid input;;
     esac
```

EXAMPLE ON CASE

```
#!/bin/bash/
 echo Enter two numbers
read num1
read num2
echo enter your choice A For Addition, S for Subtraction,D for Division, M for Multiplicatio
 read choice
 ase $choice in
  a|A)echo `expr $num1 + $num2` ;;
  s|S)echo `expr $num1 - $num2` ;;
  d|D)echo `expr $num1 / $num2` ;;
  m|M)echo `expr $num1 \* $num2`;;
      *)echo The choice is wrong;;
 esac
 echo End of CASE program
```

EXAMPLE

```
#!/bin/bash
read number
case $number
1)    echo 1st
      break;;
2)    echo 2nd
      break;;
3)  echo 3rd
     break;;
*) echo ${number}th
    break;;
esac
```

FUNCTIONS

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a "regular" command.

Shell functions are executed in the current shell context; no new process is created to interpret them. Functions are declared using this syntax:

[function] name () { command-list; }

FUNCTIONS

- Shell functions can accept arguments
- Arguments are passed in the same way as given to commands
- Functions refer to arguments using $1, $2 etc., similar to the way shell scripts refer to command line arguments

FUNCTIONS

Function to convert standard input into upper case

```
toupper()
{
    tr  “[a-z]”  “[A-Z]”
}
```

This function can be used as

```
$  cat abc | toupper
```

DEBUGGING SHELL SCRIPTS

- Two options help in debugging shell scripts
  - "-v" (verbose) option:

    causes shell to print the lines of the script as they are read.

    $ bash –v script-file

  - "-x" (verbose) option:

    prints commands and their arguments as they are executed.

    $ bash –x script-file

QUIZ

1.Which is the earliest and most widely used shell that came with the UNIX system?

A. C shell
B. Korn shell
C. Bourne shell
D. Smith shell

Option C

2.Function to convert standard input into upper case?

A. Convert
B. To upper
C. Upper
D. case

Option B

QUIZ

3.The option that causes shell to print the lines of the script as they are read?

A. -v

B. -x

C. -r

D. -w

Option A

4.prints commands and their arguments as they are executed.

A. -r
B. –w
C. –x
D. -v

Option C

PROGRAMMING IN C VS SHELL

Comparison between

- A solution in C
- A shell solution written like a C program
- A proper “shell/unix” solution

e.g:

The problem is to count the no of lines in a file (the file is called the_file)

A Shell Solution

```
#include <stdio.h>
void main(void)
{
    int lcount=0;
    FILE *infile;
    char line[500];
    infile=fopen("the_file","r");
    while(!feof(infile))
    {
        fgets(line,500,infile);
        lcount ++;
    }
    printf("no of lines is %d\n",lcount);
```

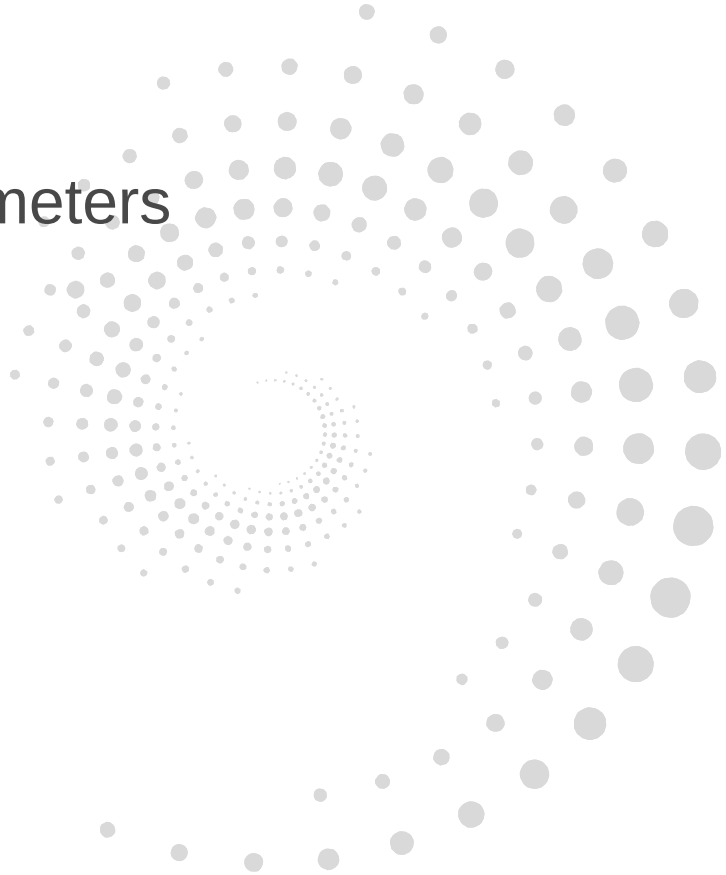
```
count=0
while read line
do
  count=`expr $count + 1`
done < the_file

echo Number of lines is $count
```

Solution using existing commands

$ wc –l the_file

SUMMARY

- **In this session, you have learned to:**
  - Use Shell variables
  - Write scripts to process positional parameters
  - Use “test” command
  - Use “if” construct
  - Use “for” loop
  - Use “while” loop
  - Use “case” construct
  - Define and use functions
  - Debug shell scripts



UNIX/LINUX

MODULE 7: ADVANCED SCRIPTING IN LINUX (EGREP, SED AND AWK)

ADVANCED FILTERS

- egrep, fgrep
- sed
- awk

EGREP , FGREP

- **Egrep**
  - Extended Grep used to search for more than one patterns at a time
  - used as egrep ‘pattern1|pattern2’ filename
  - Options used

    n - prints the line numbers

    c - count of lines matching the pattern

    v - inverses the output of the search
- **fgrep** - similar to the egrep command
  - Alternate patterns are seperated by a new line character.
  - Usage:

    $fgrep ‘a

    >b ‘ filename

SED

- sed is a stream editor

- Reading lines using sed

Examples:

sed ‘4q’ file	read first 4 lines of the file and quit
sed –n ‘1,4p’ file	print line 1 to 4
sed –n ‘1,2!p’ file	print the expect the first two lines
sed ‘G’ file	adds a blank line after each line
sed ‘4G’ file	adds a blank line after 4th line.

“ SED- SUBSTITUTION ”

- “s” for substitution

Syntax:

- sed 's/old/new/' <file_ip >file_op

- There are four parts to this substitute command:
  - s Substitute command
  - /../../ Delimiter
  - old Regular Expression Pattern String
  - New Replacement string

"SED- SUBSTITUTION "

- If you want to change a pathname that contains a slash, you could use the backslash to quote the slash
  - sed 's/\/usr\/local\/bin/\/common\/bin/' <file_ip >file_op
  - sed 's_/usr/local/bin_/common/bin_' <file_ip >file_op

```
# substitute (find & replace) "foo" with "bar" on each line
 sed 's/foo/bar/'          # replaces only 1st instance in a line
 sed 's/foo/bar/4'         # replaces only 4th instance in a line
 sed 's/foo/bar/g'         # replaces ALL instances in a line

 # substitute "foo" with "bar" ONLY for lines which contain "baz"
 sed '/baz/s/foo/bar/g‘
 # substitute "foo" with "bar" EXCEPT for lines which contain "baz"
 sed '/baz/!s/foo/bar/g'
```

"SED –N /P /W"

sed '/root/p' /etc/passwd (note line with "root" written twice)

- "sed -n," argument will not, by default, print unmodified lines.

  sed –n '/root/p' /etc/passwd (note difference w.r.t above command)

- When the "-n" option is used, the "p" flag will cause the modified line to be printed.
  - sed –n '/root/p' /etc/passwd (acts as grep)

- With /w filename you can specify a file that will receive the modified data
  - sed –n '1,4pw /tmp/op_file' /in_file (copies first four lines from in_file to op_file)

"SED MULTIPLE COMMANDS"

- Below command replaces "red" with "RED" and "white" with "WHITE"
  - sed –n 's/red/RED/gp' test | sed 's/white/WHITE/g'
  - $ sed 's/5600/5555/gp' emp.dat |sed 's/2000/2222/g'
  - sed –e 's/red/RED/p' –e 's/white/WHITE/q' test

- sed –f scriptname:

Syntax: sed –f sedscript < ip_file

- If you have (multiple) large number of sed commands you can create sed command file and execute with –f option

Example: Edit a file "sedcmd" and add the lines below

s/red/RED/g

s/white/WHITE/g

and execute it as follows

sed –f sedcmd <test_ip >test_op

“SED”

- you could remove the encrypted password from the password file (“:” acts as delimiter for input stream)
- *sed 's/[^:]*//2' </etc/passwd >/etc/password.new*
- If you wanted to add a colon after the 80th character in each line, you could type (“.” for every character)
- *sed 's/./&:/80' <file >new*
- If you want remove the comment lines
- *sed ‘/^#/d’ filename*
- To count number of lines in a file
- *sed –n ‘$=‘ filename*
- With “&” you can put the string you found in the replacement string

sed -n 's/pattern/&/p' <file (acts as grep)

"QUOTING SED IN SCRIPT"

- **Example: Reverse the order of the lines in the file and removes blank lines**

```
#!  /bin/ksh

sed '

# 1!G;h;$!d

/./!d' < filename
```

"SED ADDING RECORDS"

- **Records can be appended, inserted, deleted as below**

Example1:

```
sed 'a\
LINE 1 ADDED\
LINE2 ADDED' test
```

Example2:

```
sed `/orange/ a\
ABOVE LINE HAS orange IN IT ' test
```

```
sed '3i\
Insert a  line at line 3\
' test
```

"SED DELETING RECORDS"

Example:

sed '2d' file: delete 2nd line of the file

sed '1,4d' file: delete first 4 lines of the file

sed '1,4!d' file: delete all other lines except first four

sed 's/^[\t]*//': del leading spaces and tabs at the front of each line

sed 's/[\t]*$//': del trailing spaces and tabs at the end of each line

QUIZ

1.Which is the earliest and most widely used shell that came with the UNIX system?

A. C shell
B. Korn shell
C. Bourne shell
D. Smith shell

Option C

2.Which symbol will be used with grep command to match the pattern pat at the beginning of a line?

A. ^pat
B. $pat
C. pat$
D. pat^

Option A

QUIZ

3.The command used for modifying the files in unix ?

A. chmod

B. fgrep

C. awk

D. Sed

Option D

4.command used to search one or more files for lines that match the given string or word?

A. fgrep

B. egrep

C. awk

D. sed

Option A

Q & A

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

People matter, results count.

About Capgemini

With more than 125,000 people in 44 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2012 global revenues of EUR 10.3 billion.

Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model

www.capgemini.com/bim