Chapter 4 - Shell and Shell Programming

Introduction

- The shell sits between user and the operating system, acting as a command interpreter.
- It reads user terminal input and translates the commands into actions taken by the system.
- The shell is analogous to ***command.com*** in DOS.
- When you log into the system you are given a default shell.
- When the shell starts up it reads its startup files and may set environment variables, command search paths.
- The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.
- It has very good features for controlling input and output, but is not well suited for the interactive user. To meet the latter need the C shell, ***csh***, was written and is now found on most, but not all, Unix systems.
- It uses C type syntax, the language Unix is written in, but has a more awkward input/output implementation.
- The default prompt for the Bourne shell is **$** (or #, for the root user). The default prompt for the C shell is **%**.
- Numerous other shells are available from the network. Almost all of them are based on either ***sh*** or ***csh*** with extensions to provide job control to ***sh***
- Some of the more well known of these may be on your favorite Unix system: the Korn shell, ***ksh***, by David Korn and the Bourne Again Shell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***.
- The C shell, the Korn shell and some other more advanced shells, retain information about the former commands you've executed in the shell.

SHELL'S FUNCTIONS

- Program execution
- Name substitution and gobbling
- Input/output redirection and pipes
- Environment control.
- Programming language

The sh command:

- When you log in, you get $ prompt. As long as you do not enter something, the prompt remains idle. But, the UNIX command sh is running.
- The shell sh starts running the moment you log in. sh command can be located in /bin. sh is waiting for you to enter command and the system is idling.
- The shell swings into action the moment you enter something through the keyboard. The shell, technically the UNIX command itself, it accepts other UNIX commands as input.
- The shell first sees whether the command line is in the form the kernel can understand. If it is not, it processes the request to recreate a simplified command line.
- It then leaves the job of command execution to the kernel.
- The kernel directly communicates with all hardware and takes care of all simple or complex processes. Since the kernel interacts directly with the shell, the user remains transparent to the complex internal processes that take place between the kernel and hardware.
- The major time spent by the shell is in waiting for the input from the user. With this input, it performs series of processing tasks. It interacts with the kernel if required. After the job is complete, it returns to its waiting role, to start the next cycle.
- The following activities are typically performed by the shell in each cycle-

  1. It issues $ prompt and waits for user to enter a command

2. After the command has been entered, the shell scans the command line for some special characters and then rebuilds the command line after processing is complete.

3. The command is then passed to kernel for execution and the shell waits for its completion.

4. The $ prompt appears and the shell waits for new command.

- When there is no input from the user, the shell is said to be sleeping. It indicates this by $ prompt; which indicates that it is ready to accept new command. It wakes up whenever user enters some input and presses enter key. Sleeping, waiting and waking are the UNIX terms used for this.

Pattern Matching- the wild cards

- Metacharacters are the characters having special meaning to the shell. They should not be part of filenames.
- e.g. < > | ; ! ? * [ ] $ \ “ ‘ ` ~ ( ) { }
- The Metacharacters related to file names are known as wild cards
- ? * [ ] [-]

The ? and * :

- ?: used for single character matching
- $ ls ne?

  net new
- $ls –x chap?

  chapx chapy chapz
- $ls –x chap??

  chap01 chap02 chap03 chap04
- The wild card ? Is expanded by the shell to match any single character in a file name.
- It can't match ‘.’ as a first character of a file name.
- ‘*’ : used for multiple character matching
- $ls –x chap*

chap01 chap02 chap03 chap04 chapx chapy

- $ls –x *chap*

  chap newchap chap01 chap02 chap03.txt

- ‘*’ is expanded by the shell to match zero to any number of characters in a file name.

The shells wild cards-

- ?: matches any single character in a file name.
- *: matches multiple characters in a file name, including none.

The character class:

  - [abc]: matches a single character either a, b or c
  - [!abc]: matches a single character that is not a,b or c
  - [a-g]: matches a single character in the range a to g
  - [!a-g]: matches a single character not in the range a to g

- The patterns formed with ? & * are general and are not very restrictive. To frame very restrictive patterns, the shell provides the character classes.
- The character class is represented by a pair of square brackets. Any number of characters can be enclosed in it, but only one character is used for matching.
- The position held by brackets and their contents will be expanded by the shell to be one single character. The character will either be a member of list/range or NOT be a member of list/range.
- $ls –x chap0[124]

  chap01 chap02 chap04

- $ls –x chap[1-4]

  chap01 chap02 chap03 chap04

- $ls –x ne[stw]

  net new

- $ls –x *[1-3]

Negating the class: (!)

- Placing ! at the beginning of class reverses the matching criterion i.e. it matches all other characters except those included in the class.
- $ls –x chap0[!y-z]: it lists all files which begin with chap0 and do not end with character y & z.
- $ls –x [!a-zA-Z]* : it lists all files which do not begin with alphabet
- Wild cards mean nothing to a command, but only to shell. The shell interprets the wild cards and produces a clean list of filenames, which is easily understood by kernel.

- When wild cards lose their meaning-
- Some of the wild card characters have different meaning depending on where they are located in the pattern-

  *, ?:lose meaning when used inside class and matched literally

  -, !: lose their significance when placed outside the class

  !: lose meaning when placed anywhere except at the beginning of class

  -: lose meaning if not bounded properly on both sides by a single character

Matching a dot-

The * does not match all files beginning with a '.' or / of a pathname. Such files must be matched explicitly.

```
$ls –x .???*
```

However, if a filename contains a dot anywhere except at the beginning, it need not be matched explicitly.

```
$ls –x emp*lst
emp.lst  emp1.lst  emp22.lst emp2.lst
```

Using rm with *:

- $rm chap*
- $rm chap * # Be cautious when using rm with *

Escaping the backslash \:

- Metacharacters should not be used in filenames.
- If they are used, they cause lot of trouble.
- Assume that a file with name chap* exists.

  $ls –x chap*

  chap chap* chap01 chao02

  This file chap* can be of great nuisance value.

  Guess what happens with the following commands!

  $cat chap*

  $rm chap*
- To remove such files, the shell uses another special character \, to change the special meaning of any meta character placed after it.
- Here the shell should be informed that * has to be treated and matched literally, instead of being treated as a meta character.
- $ls –x chap* #matches * literally

  chap*

  $rm chap*

  $ls –x chap*

  chap* not found

Escaping the <enter> key :

When you enter a long chain of commands, you can split the command line by hitting enter key, after \ key

$wc –l chap04 note \ <enter>

>unit01 <enter>

The appearance of a second prompt (>) in the command line indicates that the command is not complete.

Quoting:

- Escaping is not a convenient solution when you need to despecialize the meaning of group of characters.
- When an argument to a command is enclosed in quotes, the meaning of all special characters is turned off.
- e.g.

  $echo * : displays *

  $echo '*': displays *
- When you have a continuous set of metacharacters to be despecialized, quoting is preferable to escaping.

Quoting preserves spaces:

- $echo C W I T PUNE

  CWIT PUNE
- $echo "C W I T PUNE"

  C W I T PUNE

Escaping in echo

- \ is also used to emphasize a character so that a command treat it as a special character.
- $ echo 'Enter your name : \c'

  Enter your name :$

  Echo also accepts certain escape sequences such as \t, \f, \n

  $echo '\tThis message is broken here\n \ninto three lines'

  This message is broken here

  ---

  into three lines

Redirection:

- The family of commands take input from a file and send output to the terminal. These commands can also operate on character streams(sequence of bytes). A command can be ignorant about source and destination of these streams. The shell sees that these streams are handled properly.
- These commands can be made to accept input from other sources and to send output to some other destination than terminal.

Standard Input:

The command considers its own input as a stream. This stream can come from-

1) the keyboard (default)

2) a file (using redirection)

3) another program (using pipeline)

- eg cat and wc commands need filename(s) as an argument. But when a file name is missing, the input has to be keyed in –

  1) $wc

  C W I T PUNE

  Computer Dept

  <cntrl z> <enter>

  2) You can take the standard input from a file, instead of keyboard.

  $ wc < infile

  --- --- ----

  Here the connection has been set up by the shell using metacharacter <

  The shell has redirected the stream or reassigned the standard input file to come from a disk file. wc has no idea of where the stream came from. Also, the shell has to open the infile and take input from there.

  3) $ wc infile

  --- --- ---- infile

  File is opened by the command wc itself.

Standard Output:

Like the standard input, the standard output stream also has three similar destinations-

1. Output can be directed to terminal (default)
2. Output can be directed to file(using redirection)
3. Output can serve as a input to another program.

- Any command displaying output on the screen is normally using standard output. The mechanism of redirection is also available with standard output.
- You can replace the default destination of any file using > operator, followed by a file name.

  $ wc infile > newfile

  $ cat newfile

  3 20 103 infile

  This sends output of wc to a file newfile. If the newfile does not exists, it is created. If it exists, it is overwritten.

Combining standard input and standard output:

- The standard input and standard output can be combined in the same command.
- When using the default source i.e. keyboard, just use – and > (or >>) symbols with command.
- cat > file1 and cat -> file1 are same.
- To append to the file,

  cat >> file1 and cat ->> file1

  In this editing mode you can not see the original contents of file.

  A command can use both, the standard input and standard output, the < and > operators can be combined to use both forms of redirection in a single command line.

  $wc < infile > newfile

Many important UNIX commands make use of redirection.

The ignorance of a command to the source of its input and destination of its output is one of the important features of UNIX.

Therefore, the output of one command can be input to another. Such commands are called as filters.

Standard Error:

- The third stream, standard error includes all error messages written to the terminal. This output may be generated by command or by the shell, but in either case the default destination is terminal. This output can also be reassigned to a file.

  $cat bar > errorfile

  cat : can not open bar: no such file or directory

  The system still displays output on the terminal, the error massage is not directed to the errorfile.

File descriptors:

- Each of these three standard files has a number associated with them, called File descriptor, which is used for identification.

0: represents standard input

1: represents standard output

2: represents standard error

- The File descriptor 2 is required to be prefixed to the redirection symbol.

  $cat bar 2> errorfile

  $cat errorfile

  cat : can not open bar: no such file or directory

Pipes:

- The standard input and standard output constitute two separate streams, which can be manipulated separately by the shell. The shell can also connect these streams together, so that one command takes input from the other. This is important features of UNIX.
- eg who : displays list of users, one user per line
- To count the number of users-
- $who > user.lst
- $cat user.lst

  abc tty0 April 18 10:45

  pqr tty1 April 18 10:49
- $wc-l user.lst

  2 user.lst

  Disadvantages of this method-

  i. you need intermediate files
  ii. process is slow.
  ii. for complex tasks not convenient
  iv. consumes disk space

The shell enables the standard output of one command to be connected as (std) input to another. The | pipe symbol is used as a connector of commands.

- Since *who* sends out a character stream, and *wc* accepts it, the above sequence of commands can be combined into a single instruction as-

  $who | wc –l <enter>

  2

  Here the output of *who* has been passed directly to the input of *wc* and *who* is said o be piped to *wc.* No intermediate files are created.

  When a sequence of commands are combined together, a pipeline is said to be formed.

  $ls | wc –l : counts number of files in a current

  15 directory

tee:

- UNIX provides a feature by which you can save the standard output in a file as well as display it on terminal(or pipe it to a another process). This is made possible by the tee command, which is available in /bin.
- Tee uses standard input and standard output, so that it can be placed anywhere in the pipeline.
- Tee breaks up the input into two components –one component is saved in a file and the other is connected to the standard output. Tee performs no filtering action.
- You can use tee to save the output of *who* command in a file, as well as display it.

  $who | tee user.lst

  abc tty0 April 18 10:45

  pqr tty1 April 18 10:49

  One channel is saved in user.lst, while the other is displayed on the standard output.

  Since the tee uses standard output, you can pipe its output to another command, say, *wc*

  $who | tee user.lst | wc –l

  2

Command Substitution:

- UNIX allows to connect two commands in different ways. Pipe is one method.
- The shell allows the argument of a command to be obtained from the output of another command. This feature is called Command Substitution.
- When a command is enclosed in a pair of back quotes, the shell executes the command first, and the enclosed command text is replaced by the output of the command. It is like running a command within command.

- $echo The todays date is ; date

  The todays date is Wed April 18 1:25:15 IST 2020
- $echo The todays date is `date`

  The todays date is Wed April 18 1:25:15 IST 2020
- When scanning a command, if the shell comes across with the pair of ` metacharacters, the shell executes the enclosed command and puts the output in its place.
- For the Command Substitution to work, enclosed command must use standard output.
- $echo There are `ls |wc –l` files in current directory

  There are 58 files in current directory
- The back quote does not lose its meaning when placed within double quotes.

Shell Variables:

The shell provides the facility to define and use variables in the command line. These variables are called shell variables. No type declaration and initialization are required before its use.

Shell variables are assigned with = operator and evaluated by prefixing the variable name with $

- $ x=37
- $echo $x

  37
- All shell variables take the general form-

  variables = value
- $ MyVar=999
- $ echo $MyVar

  999
- $ echo "$MyVar"

  999

- $ echo '$MyVar'

  $MyVar
- $ city=Delhi
- $ echo "We are in $city today."

  We are in Delhi today.
- $ echo 'We are in $city today.'

  We are in $city today.

- Shell variables are of string type i.e. the value is stored in ASCII rather than binary. The shell interprets any word preceded by $ as a variable and replaces the word by the value of variable. All shell variables are initialized to null string by default. An unassigned variable returns a null string.
- $echo $xyz

  $-

  The null strings can be assigned explicitly as

  X= or X=‘ ‘
- To assign multi word string, quotes are used as

  $ msg=‘You have Mail’

  $echo $msg

  You have Mail.

Ch4: Shell Programming

Introduction

- The UNIX shell supports programming in addition to its other functions.
- The shell has a set of internal commands that can be stringed together as a language, with its own variables, conditionals and loops.
- Shell programs run in interpretive mode i.e. one statement at a time. Therefore, shell programs run slower than those written in high level languages.

Shell Script

- All shell statements and UNIX commands can be entered in the command line.
- When a group of commands has to be executed regularly, they can be stored in a file.
- All such files are called shell scripts, shell programs or shell procedures.
- These files can have any name and may have .sh extension.
- Consider the following script-

```
$ cat script.sh        # Sample shell script
date                   # Use # to comment lines
cal 'date "+%m 19%y"‘
echo 'Calendar for the current month shown above'
```

- You can use the vi editor to create this script. You can execute this file in two ways. One way is to use the sh command along with the filename:

```
sh script.sh
```

- Alternatively, you can first use chmod to make the file executable, and then run it by simply invoking the filename:

  $ chmod +x script.sh

  $ script.sh

  Displays the calendar of current month

- The three statements have been executed in sequence. Comment (beginning with #) can be placed anywhere in a line.

read: Making Scripts Interactive-

- The read statement is the shell's internal tool for taking input from the user, i.e. making scrip interactive
- It is used with one or more variables, and input supplied through the standard input is read into these variables.
- The script empl.sh uses read to take a search string and filename from the terminal:

  $ cat emp1.sh

  # Script: empl.sh - Interactive version

  # The pattern and filename to be supplied by the user

  echo "\nEnter the pattern to be searched: \c“

  read pname

  echo "\nEnter the file to be used: \c"

  read flname

  echo "\nSearching for $pname from file $flname\n"

  grep "$pname" $flname

  echo "\nSelected records shown above\n"

- The script empl.sh uses read to take a search string and filename from the terminal:

```
$ cat emp1.sh
# Script: empl.sh - Interactive version
# The pattern and filename to be supplied by the user
echo "\nEnter the pattern to be searched: \c“
read pname
echo "\nEnter the file to be used: \c"
read flname
echo "\nSearching for $pname from file $flname\n"
grep "$pname" $flname
echo "\nSelected records shown above\n"
```

- The script pauses at two points. First, it asks for a pattern to be entered. Input the string "di rector", which the shell assigns to the variable pname. Next, it asks for the filename; enter the string "emp2.lst", which goes to the variable flname. After accepting these inputs from the keyboard, the grep and echo statements are executed.
- A single read statement can be used with one or more variables to let you enter multiple arguments:

```
read pname flname
```

Command line arguments-Positional Parameters

- Shell procedures accept arguments in the command line itself.
- This non-interactive method of specifying arguments is quite useful for scripts requiring few inputs.
- It forms the basis of developing tools that can be used with redirection and pipelines.
- When arguments are specified with a shell procedure, they are assigned to certain special "variables", or *positional parameters.*

- The first argument is read by the shell into the parameter $1, the second argument into $2, and so on.
- There are a few other special parameters used by shell. The next script illustrates these features:

  $ cat emp2.sh

  echo "program: $0 # $0 contains the program name

  The number of arguments specified is $#

  The arguments are $*“ # All arguments stored in $*

  grep "$1" $2

  echo "\nJob Over"
- When arguments are specified in this way, the first word (the command itself) is assigned to $0, second word (the first argument) to $1, and the third word (the second argument) to $2. You use more positional parameters in this way up to $9
- Since the script accepts three arguments, how do you search for a multi-word patterns "barun sengupta" from emp2.lst?
- When the string is quoted, the shell understands it as a single argument:
- $ emp2.sh "barun sengupta" emp2.lst
- Program: emp2.sh
- The number of arguments specified is 2
- The arguments are barun sengupta emp2.lst
- 2365|barun sengupta|director|personnel| 11/05/47|7800
- Job Over
- *$#* has been set at 2. If you had not used quotes, then the line would have been printed twice, first for "barun", and then for "sengupta".

exit Status of A Command

- Every command *returns* a value after execution. This value is called the *exit status* or *return value* of the command. A command is said to be true if it executes successfully, and false if it fails. eg-

  $ cat foo

  cat: cannot open foo

  generates an error message to the standard error.

- The cat command shown above didn't succeed, & is thus said to return a false exit status.
- These return values are of great importance to the programmer who has to devise program logic that branches into different paths, depending on the success or failure of a command.
- The shell offers a number of operators and statements that test these values and make use of them.

The Parameter $?

- The parameter $? stores the exit status of the last command. It has the value 0 if the command succeeds, and a non-zero value if it fails.
- For example, if grep fails to find a pattern, the return value is 1, and if the file scanned is unreadable in the first place, the return value is 2.

  $ grep director emp.lst

  $ echo $?

  0

  $ grep manager emp.lst

  $ echo $?

  1 # *manager doesn't exist*

  $ grep manager emp3.1st

  grep: can't open emp3.1st

$ echo $?

2

- All assignments to the positional and special parameters are automatically made by shell.

Special Parameters Used by the Shell

Shell parameter	Significance
1. $1, $2, etc.	The positional parameters
2. $*	Complete set of positional parameters as a single string
3. $#	Number of arguments specified in command line
4. $0	Name of executed command
5. $?	Exit status of last command
6. $!	PID of last background job
7. $@	Same as $* except when enclosed in double quotes

The logical operators && and ||-conditional execution

- The shell provides two operators to control execution of a command depending on the success or failure of the previous command the && and ||
- The && operator is used by the shell to combine two commands; the second command is executed only when the first succeeds.

  $ grep 'director' empl.1st && echo "pattern found in file"

  1006|chanchal singhvi |director |sales|03/09/38|6700

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

  pattern found in file
- The || operator is used to execute the command following it when the previous command fails. If you "grep" a pattern from a file without success, you can notify the failure:

$ grep 'manager' emp2.lst II echo "Pattern not found”

Pattern not found

exit: Script Termination

- The exit statement is used to prematurely terminate a program. When this statement is encountered in a script, execution is halted and control is returned to the calling program, in most cases the shell.

  $ grep "$I" $2 || exit 2 # exit also takes an argument

  echo "Pattern found - Job Over“
- The program is aborted when the grep command fails, and a message is printed if it succeeds.
- The argument provided with exit is optional. When you specify one, the script will terminate with a return value of the argument. If no return value is specified, the value returned will be zero (true). This value is assigned to the parameter $?

Conditional Statements

The if- Conditional

The general form is

if *condition* is *true*

then

execute commands

else

execute commands

fi

if *condition is true*

then

execute commands

fi

- The condition can be simple one like comparing two values, *or checking the return value of any UNIX program.* All UNIX commands return a value – true or false

```
$ if grep "director" emp.lst
> then echo "Pattern found - Job Over"
> else echo "Pattern not found“
> fi
9876 | jai Sharma | director | productionI 12/03/90| 70000
2365 | barun guptaIdirector I personnel 11/05/87| 78000
1006 | chanchal singh I director| sales | 03/09/98| 67000
6521 | lalit kale I director | marketing | 26/09/95| 82000
 Pattern found - Job Over
```

Numeric Comparison with test

- When if is used to evaluate expressions, the test statement is often used as its control command. test uses certain operators to evaluate the condition on its right, and returns either a true or false exit status, which is then used by if for taking decisions.
- Numeric comparison in the shell is confined to *integer* values only
- The relational operators also have a different form when used by test.
- They always begin with a - (hyphen), followed by a two character word, and enclosed on either side by white space.
- *Operators*

  -eq -ne -gt -ge -lt -le
- test doesn't display any output, but simply returns a value, which is assigned to the parameter S?
- $ x=5; y=7; z=7.2
- $ test $x -eq $y ; echo $?

  1
- $ test $x -lt $y ; echo $?

0

- $ test $x -gt $y ; echo $?

  1

- $ test $z -eq $y ; echo $?

  0

- The next script uses three arguments to take a pattern, as well as the input and output filenames. First it checks whether the right number of arguments are entered

- $ cat emp3. sh

```
if test $# -ne 3     # If 3 arguments are not entered then
   echo "You have not keyed in 3 arguments"
   exit 3
else
   if grep "$1" $2 > $3     then
            echo "Pattern found - Job Over"
   else
            echo "Pattern not found - Job Over"
   fi
fi
```

```
$ emp3.sh manager emp.lst man.lst
Pattern found - Job Over
$ cat man.lst
1265|s. n. dasgupta| manager|sales | 12/09/63|5600
2345|anil aggarwal|manager|sales|01/05/59|5000
```

Short-hand for test:

- test is widely used, so a shorthand method is provided for executing it. A pair of rectangular brackets enclosing the expression can replace the word test.

  test $x -eq $y or [$x -eq $y]

  [and] to have spaces on their inner sides

test: String Comparisons:

- test can be used to compare strings with yet another set of operators. Equality is performed with =, the operator != checks for inequality. Like the other test operators, these also should have white space on either side.

	test	Exit Status
1.	-n *stg*	True if string s*tg* is not a null string
2.	-z *stg*	True if string *stg* is a null string
3.	s1 = s2	True if string s1 = s2
4.	s1 != s2	True if string s1 is not equal to s2
5.	stg	True if string *stg* is assigned and not null

- $ cat emp4.sh

```
echo "Enter the string to be searched: \c"
read pname
        i f [ -z "$pname" ] ; then  # -z checks for a null string
        echo "You have not entered the string" ; exit 1
else
        echo "Enter the file to be used: \c"
read flname
        i f [ ! -n "$fl name" ] ; then      # ! -n is the same as -z
        echo "You have not entered the filename" ; exit 2
```

```
else
        grep "$pname" "$flname" IIecho "Pattern not found"
fi
fi
```

test: File Tests

test can be used to test the various file attributes. For example, you can test whether a file has the necessary read, write or executable permissions. Any of the test options can be negated by the ! (bang) operator. Thus, [! - f file] negates [-f file].

The file testing syntax used by test is simple, and you can test some attributes of the files at the prompt:

- $ ls -l emp.lst

  -rw-rw-rw- 1 kumar group 870 Jun 8 15:52 emp.lst
- $ [-f emp.lst] ; echo $?

  0 *# An ordinary file*
- $ [-x emp.lst] ; echo $?

  1 *# Not an executable*

THE case CONDITIONAL

The statement matches an expression for more than one alternative, and uses a compact construct to permit multi-way branching. It also handles string tests. The general syntax of the case statement is as *follows:*

```
case expression in
   patternl) execute commands ;;
   pattern2) execute commands ;;
   pattern3) execute commands ;;
esac
```

case matches the *expression* first for *patternl,* and if successful, executes the commands associated with it. If it doesn't, then it falls through and matches *pattern2,* and so on. Each command list is terminated by a pair of semi-colons, and the entire construct is closed with esac.

```
$ cat menu. sh
echo “        MENU\n
1. List of files\n  2. Processes of user\n
3. Today's Date\n   4. Users of system\n
5. Quit to UNIX\nEnter your option: \C"
read choice
case “$choice" in
1) ls -l ;;
2) ps -f ;;
3) date;;
4) who ;;
5) Exit         ;; not really required for the last option
esac

$ menu.sh
          MENU
  1. list of files
  2. Processes of user
  3. Today' s Date
  4. Users of system
  5. Quit to UNIX
Enter your opt ion: 3
   Fri Apr 27 11:41:16 IST 2007
```

expr COMPUTATION:

- The shell doesn't have any computing features at all; it has to rely on the expr command for that purpose. This command combines two functions in one; it can perform arithmetic operations on integers, and also manipulate strings to a limited extent.
- It can be useful in handling simple arithmetic tasks. As a computing tool, expr can perform the basic four arithmetic operations, as well as the modulus function:

  $ x=3 y=5
  $ expr 3 + 5
  8
  $ expr $x - $y
  -2
  $ expr 3 \ * 5
  15
  $ expr $y / $x
  1

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

$ x=6 y=3 ; z=‘expr $x + $y’
$echo $z
9

- Perhaps the most common use of expr is in incrementing the value of a variable

  $ x=5
  $ x=‘expr $x + 1‘
  $ echo $x
  6

while: LOOPING

- The shell features three types of loops--while, until and for. All of them repeat the instruction set enclosed by certain keywords.
- while repeatedly performs a set of instructions till the control command returns a true exit status. The general syntax of this command is as follows:

  while *condition* is *true*
  do
    execute commands
  done

for: LOOPING WITH A LIST

The for loop is different in structure from the one used in other programming languages. Unlike while and until, it doesn't test a condition, but uses a list instead. The syntax of this construct is as follows:

for *variable* in *list*
do
execute commands
done

- The loop body is the same but the additional parameters **here are** *variable* and *list.* The loop body is executed as many times as there are items in the list.
- # table of n
  X=1
  Y=21
  K=0
  for x in list 1 2 3 4 5 6 7 8 9 10
  do
    let k=$x*$y
    echo $k
  done

Other form of for loop –

General form -

```
for(exprn1; exprn2; exprn3)
 do
        statement;
 done
```

eg.

```
for ( i=1; i<=10;  i++)
  do
       let k=$i\*9
      echo -n "$k"
   done
```

- until statement

The until loop is executed as long as the condition evaluates to false. The loop terminates when the condition becomes true.

Syntax:

```
until condition
do
   Statement/s
done
```

```
a=0
until [ $a -gt 10 ]
do
    echo $a              # Print the values
    a=`expr $a + 1`      # increment the value
done
```