UTS

UNIX PROGRAMMING (JNTUK-R16)
UNIT-3

SYLLABUS:

- *Using the Shell*
    - Command Line Structure
    - Meta characters
    - Creating New Commands
    - Command Arguments and Parameters
    - Program Output as Arguments
    - Shell Variables
    - More on I/O Redirection
    - Looping in Shell Programs

Text Books:

1. The Unix programming Environment by Brain W. Kernighan & Rob Pike, Pearson.
2. Introduction to Unix Shell Programming by M.G.Venkateshmurthy, Pearson.
3. UNIX and Shell Programming, Forouzan and Gilberg, Cengage Learning

1. COMMAND LINE STRUCTURE:

- A **command** is a program that tells the Unix system to do something.

Syntax:

```
command [options] [arguments]
```

- Where an argument indicates on what the command is to perform its action, usually a file or series of files. R[
- An option modifies the command, changing the way it performs.
- Commands are case sensitive.

Example:

- command and Command are not the same.
- Options are generally preceded by a hyphen (-)
- For most commands, more than one option can be strung together

Syntax:

```
command -[option][option][option]
```

Example:

```
$ls -alR
```

- will perform a long list on all files in the current directory and recursively perform the list through all sub-directories.
- For most commands you can separate the options, preceding each with a hyphen

Syntax:

```
command –[option1] –[option2] –[option3]
```

Example:

```
$ls -a -l –R
```

- Options and syntax for a command are listed in the man page for the command.

2. UNIX META CHARACTERS:

- These are also called as Special Characters or Wildcard Characters.
- Special characters have a special meaning to the shell.
- They can be used to specify the name of a file without having to type out the file's full name
- The most commonly used Meta Characters in UNIX are,

i) Asterisk *
ii) Question Mark ?
iii) Brackets [ ]
iv) Hyphen -

i) Asterisk: *

- The * (asterisk) meta character is used to match any and all characters.

Example:

- List all files in the working directory that begin with the letter s regardless of what characters come after it.

```
$ ls s*
```

- The * (asterisk) meta character can be used anywhere in the filename.
- It does not necessarily have to be the last character.

ii) Question Mark: ?

- The ? (question mark) meta character is used to match a single character in a filename.

Example:

- List all files in the working directory that has a single character, or two characters or three characters

```
$ls ?
$ls ??
$ls ???
```

- List all files in the working directory that has a three characters start with k.

```
$ls k??
```

- Like the asterisk, the ? (question mark) meta character can be used as a substitution for any character in the filename.

iii) Brackets: [ ]

- Brackets ([...]) are used to match a set of specified characters.
- A comma separates each character within the set.

Example:

- List all files beginning with "a", "b", or "c".

```
$ ls [a,b,c]*
```

- It is also represented as,

```
$ls [abc]*        # without giving comma seperator
```

iv) Hyphen: -

- Using the - (hyphen) metacharacter within [ ] (brackets) is used to match a specified range of characters.

Example:

- List all files beginning with a lowercase letter of the alphabet

```
$ ls [a-z]*
```

- If there are directory names in the specified range, their name and contents will also be displayed

3. I/O REDIRECTION:

- It is possible to change the source from where the input is taken by a program as well as the destination to where the output is sent by a program.
- This mechanism of changing the input source and/or destination is called Redirection.
- The UNIX Redirection Operators are

Symbol	Name	Redirection
<	Less than	Standard Input Redirection
>	Greater than	Standard Output Redirection
>>	Double Greater than	Standard Output Redirection with appending

- The input source is redirected using the < (less than) operator.
- The output destination is redirected using the > (greater than) or >> (double greater than) operators.
- The file descriptors 0 and 1 are implicitly prefixed to the redirection operators < and > respectively by the shell.
- The output of a program can be redirected using either > or >> operator.
- When > is used, the destination files are overwritten.
- When >> is used, the present output will be appended to an existing file.
- In either case if the destination file does not exist, it is created.

Example:

- Creating a new file called sample1 using >

```
$cat  >  sample1
Hi Hello
```

- By using ctrl+d command, we come out from cat editor.

- Displaying contents of sample1 using <

```
$cat  <  sample1
Hi Hello
```

- Appending the sample1 file contents using >>

```
$cat  >>  sample1
We are CSE
```

- Displaying modified contents of sample1 using <

```
$cat  <  sample1
Hi Hello
We are CSE
```

- Redirecting the contents of sample1 to another file called sample2

```
$cat  <  sample1  >  sample2
$cat  <  sample2
Hi Hello
We are CSE
```

4. CREATING NEW COMMANDS:

- When a set of commands are required to be repeated for specific number of times, then it would be better to make them in to new command.
- They can be assigned with user defined names.
- Users can use them like regular commands.

Example1:

```
$pwd
/home/501

$echo  'pwd'  >  sample         #Redirecting pwd command to a file called sample

$cat  sample
pwd

$sh  <  sample                  #Executing sample file like pwd command using sh command
/home/501
```

Example2:

```
$pwd
/home/501

$echo  'pwd'  >  sample         #Redirecting pwd command to a file called sample

$cat  sample
pwd

$mkdir bin                      #Creating bin directory in current directory

$echo $PATH                     #Setting path using the $PATH system variable
/home/501/bin

$mv sample bin                  #Moving sample file to bin directory

$cd bin

bin] $sample
permission denied               # Because there is no execution permission to the User/Owner

bin] $chmod u+x sample          #Giving Execute Permission to user using chmod command

bin] $sample                    #Executing sample command directly like pwd
/home/501
```

Example3: (As an Internal Command)

- To count number of users with pipeline frequently, the command used is

```
$who |wc -l
3
```

- An ordinary file can be created to contain 'who |wc -l'

```
$echo 'who | wc -l' > sample2
```

- The program input can be redirected to a file rather than the terminal.

```
$who
501  dev/pts/0  jul  13  10:20
502  dev/pts/1  jul  13  10:30
503  dev/pts/2  jul  13  10:40

$cat sample2
$who |wc -l

$sh< sample2
3
```

- The output is same as the initial command

Example4: (As an External Command)

- To count number of users with pipeline frequently, the command used is

```
$who |wc -l
3
```

- An ordinary file can be created to contain 'who |wc -l'

```
$echo 'who | wc -l' > sample2
```

- The program input can be redirected to a file rather than the terminal.

```
$who
501  dev/pts/0  jul  13  10:20
502  dev/pts/1  jul  13  10:30
503  dev/pts/2  jul  13  10:40

$cat sample2
$who | wc -l

$mkdir bin                  #Creating bin directory in current directory

$echo $PATH                 #Setting path using the $PATH system variable
/home/501/bin
```

```
$mv sample2 bin                 #Moving sample file to bin directory

$cd bin

bin] $sample2
permission denied               # Because there is no execution permission to the User/Owner

bin] $chmodu+x sample2          #Giving Execute Permission to user using chmod command

bin] $sample2                   #Executing sample command directly like who | wc -l
3
```

5. COMMAND ARGUMENTS AND PARAMETERS:

- The shell programs will interpret the arguments such as filenames and options while running the program.

Example:

```
        $cx  sample
```

- This command is shorthand for

```
        $chmod  +x  sample
```

- The contents of cx are

```
        chmod, +x and sample
```

- When shell executes file containing commands, every occurrence of $1 will be replaced by first argument and $2 will be replaced by second argument and so on.
- Let cx contain

```
        $chmod  +x  $1
```

- If the below command is run

```
        $cx  sample
```

- Here sub shell will replace the $1 with first argument, "sample".
- Shell can handle even multiple arguments.

```
        $chmod  +x  $1  $2  $3  $4  $5  $6  $7  $8  $9
```

- A Shorthand notation for this would be

```
        $chmod  +x  $*
```

- The argument $0 representing the command to be executed.

6. PROGRAM OUTPUT AS ARGUMENTS IN UNIX

- The shell allows the standard output of one command to be used as an argument of another command.
- The shell executes the command enclosed within single quotes and replaces the command with standard output.
- This feature is called command substitution.

Syntax:

```
        'command'
```

Example1: (Command Substitution)

```
$echo current date is ‘date’
 current date is Tue Aug 14 10:35:22 IST 2018
$
```

Example2: (Command Substitution to generate useful messages)

```
$who
  user1  /dev/pts/0  Aug 18 10:35
  user2  /dev/pts/1  Aug 18 10:35
  user3  /dev/pts/2  Aug 18 10:35
$
$echo “current users working on the system are ‘who | wc -l’ ”
  current users working on the system are 3
$
```

Example3: (Command Substitution in Shell Scripts)

```
$ cat > myscript.sh
  echo Number of users logged on to the system are ‘who | wc -l’
  echo The present Working Directory is ‘pwd’
^d
$
$sh myscript.sh
  Number of users logged on to the system are 3
  The present Working Directory is /home/501
```

7. SHELL VARIABLES:

- A Variable is a data name used to store data value.
- Variables are defined and used with a shell.
- Shell Variables are three types
    - i) System Variables
    - ii) Local Variables
    - iii) Read-only Variables

i) SYSTEM VARIABLES:

- These variables are also called as Environment Variables.
- These variables are set either during the boot sequence or immediately after logging in.
- The working environment under which a user works, depends entirely upon the values of these variables.
- These are similar to global variables.
- Represented in Uppercase letters
- The different system variables are,
    - a) PATH
    - b) HOME
    - c) IFS
    - d) MAIL
    - e) SHELL

f) TERM

a) The PATH variable:

- The PATH variable holds a list of directories in a certain order
- In this list colon (:) separate different directories

Example:

```
$echo  $PATH
 /usr/lib/qt-3.3/bin:/usr/kerboros/bin:/usr/lib/ccache:/usr/local/bin:/bin:
 /usr/bin:/home/501/bin
$
```

- When any command is given, the shell searches for its program in the directories listed in the PATH one by one
- If the program for the command is not found in any of these directories, the message "command not found" will be displayed

b) The HOME variable:

- When a user logs in, he or she will be automatically placed in the home directory.
- This directory is decided by the system administrator at the time of opening an account for a user.
- This directory is stored in the file /etc/passwd
- The value of the path of the home directory is stored in the variable HOME
- The user can know this value using the echo command

Example:

```
$echo  $HOME
 /home/501
$
```

c) The IFS variable:

- This variable holds tokens used by the shell commands to parse a string into substrings such as a word or a record into its individual fields
- The default tokens are the three white space tokens
    - Space
    - Tab
    - New line
- Because all these are non-printable characters, they can be seen or verified using the od command.

Example:

```
$echo  "$IFS"  |  od  -bc
 0000000  040  011  012  012
                \t    \n    \n
 0000004
$
```

- The option –b displays octal value of each character
- The option –c displays the character itself

d) The MAIL variable:

- This variable holds the absolute pathname of the file where user's mail is kept
- Usually the name of this file is the user's login name

Example:

```
$echo  $MAIL
  /var/spool/mail/501
$
```

e) The SHELL variable:

- This variable contains the name of the users shell program in the form of absolute pathname
- System administrator sets the default shell
- If required, user can change it
- The value of the variable SHELL may be known by using the echo command

Example:

```
$echo  $SHELL
  /bin/bash
$
```

f) The TERM variable:

- This variable holds the information regarding the type of the terminal being used.
- If TERM is not set properly, utilities like vi editor will not work

Example:

```
$echo  $TERM
  xterm
$
```

Other System Variables

The LOGNAME variable:

- The variable LOGNAME holds the user name

Example:

```
$echo  $LOGNAME
  501
$
```

The TZ variable:

- The variable TZ holds the current Time Zone

Example:

```
$echo  $TZ

$
```

The PS1 variable:

- The PS1 variable holds the primary prompt value ($)

Example:

```
$echo  $PS1
  $
$
```

The PS2 variable:

- The PS2 variable holds the secondary prompt value (>, right chevron)

Example:

```
$echo  $PS2
 >
$
```

ii) LOCAL VARIABLES:

- These variables are also called as User-defined Variables.
- These variables are defined and used by specific users.
- These variables are local to the user's shell environment.

Rules for constructing Variable Names:

- Shell variable names are constructed using only alphanumeric (alphabets and digits) characters and the Underscore (_)
- It starts with a letter.
- The variable names are case-sensitive.

Example:

SUM, sum,Sum, suM, sUm are different.

- Spaces not allowed.

Defining a Shell Variable:

- Shell variables are evaluated by prefixing the variable name with a $.

Syntax:

```
$variable=value
```

Example:

```
a) $x=20
b) $y=45.37
c) $z=sachin
d) $w="india is my country"
```

iii) READ-ONLY VARIABLES:

- These variables uses readonly( ) function.
- The values of variables which can only be read but not to be manipulated are called read-only variables.

Example:

```
$cat example
 echo Enter value for x
 read x
 echo value of x is $x
 readonly  x
 x='expr  $x+1'
 echo The value of x now is $x
$
```

Execution:

```
$sh example
  Enter value for x
  4
  value of x is 4
  example: line 5 : x : readonly variable
$
```

8, LOOP CONTROL STRUCTURES:

- It will be required to execute a set of statements repeatedly.
- Shell also has repeated executions
- Repeated executions also need decision-making
- The loop control structures in UNIX are three types
    - i) The while command
    - ii) The until command
    - iii) The for command
- All loop control structures in UNIX are entry-controlled loops.

i) The while command:

- The while is an entry-controlled loop structure.

Syntax:

```
while condition is true
  do
    set of commands that are executed repeatedly
  done
```

- The while, do, done are the keywords.
- The set of commands between do and done keywords are repeatedly executed as long as the condition remains true
- Any UNIX command or a test expression can be used as the condition

Example:

```
a=0
while [ $a  -lt  5]
  do
    echo $a
    a= $(expr  $a  +  1)
  done
```

Output:

```
0
1
2
3
4
```

ii) The until command:

- The until command behaves exactly opposite manner to the while command.

- The until command repeatedly executes the set of commands that appears between do and done keywords as long as the condition remains false.

Syntax:

```
until condition is false
  do
    set of commands that are executed repeatedly
  done
```

Example:

```
a=0
until [$a  -ge  5]
  do
    echo $a
    a=$(expr  $a  +  1)
  done
```

Output:

```
0
1
2
3
4
```

iii) The for command:

- The for command works with a set of values generally given in the form of a list.

Syntax:

```
for variable in list
  do
   set of commands that are executed repeatedly
  done
```

Example:

```
for a in 0 1 2 3 4
  do
   echo $a
  done
```

Output:

```
0
1
2
3
4
```