Unix Shell Script and Utilities

by
Palani Karthikeyan

SESSION 1 : Introduction to UNIX

Objectives

At the end of this module you will learn about:

- What Is UNIX
- History of UNIX
- Characteristics of the UNIX OS
- What is Linux
- Layered Architecture
- Kernel & Shell
- Common UNIX Flavors
- System boot up
- Virtual Console

What Is Unix

- In the simplest terms, Unix is an operating system.

- An operating system is the software that runs behind the scenes and allows the user to:
  - operate the machine’s hardware
  - start and stop programs
  - set the parameters under which the computer operates

- The most basic requirement of an operating system is that it permits the user to operate the computer.

- Unix operating systems are widely used in servers, workstations and mobile devices.

History of UNIX

- 1965 Bell Laboratories joins with MIT and General Electric in the development effort for the new operating system, Multics.

- 1969 AT&T was unhappy with the progress and drops out of the Multics project. Some of the Bell Labs programmers who had worked on this project, Ken Thompson, Dennis Ritchie, Rudd Canaday, and Doug McIlroy designed and implemented the first version of the UNIX File System on a Programmed Data Processor (PDP-7) system.

- 1973 Unix is re-written in C, new programming language developed by Dennis Ritchie.

- 1989, AT&T and Sun Microsystems joined together and developed system V release 4 (SVR4)

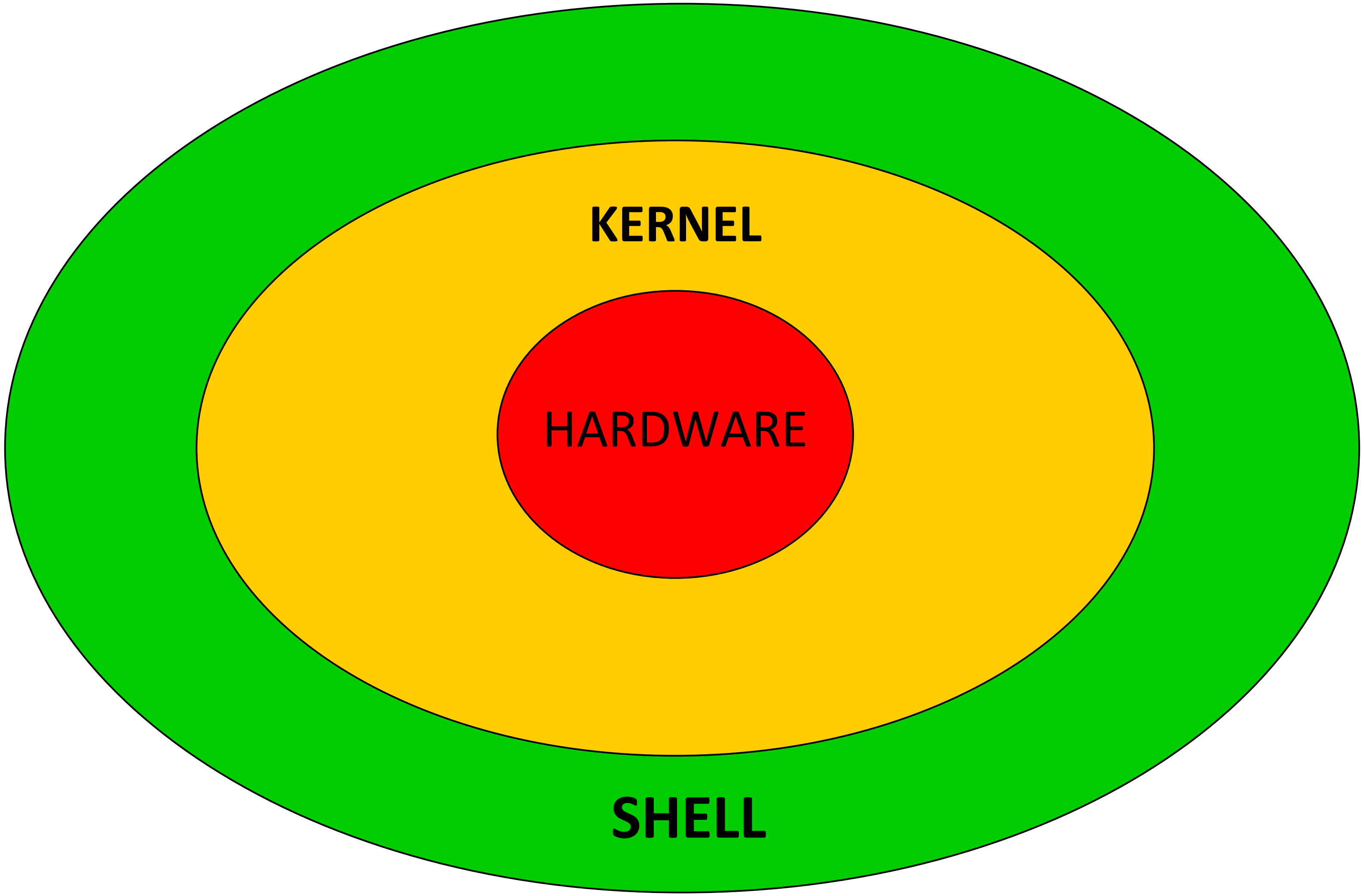
Characteristics of the UNIX OS

- Multi-user, multitasking, timesharing
- Portability
- Modularity
- File structure
- Security

What is Linux

- An open source operating system like UNIX
- Initially created by Linus Torvalds for PC architecture
- Ports exist for Alpha and Sparc processors
- Developer community world-wide contribute to its enhancement and growth

Layered Architecture



Kernel

- Kernel is that part of the OS which directly makes interface with the hardware system.

- Actions:
  - provides mechanism for creating and deleting processes
  - provides processor scheduling, memory and IO management
  - does inter process communication.

Shell

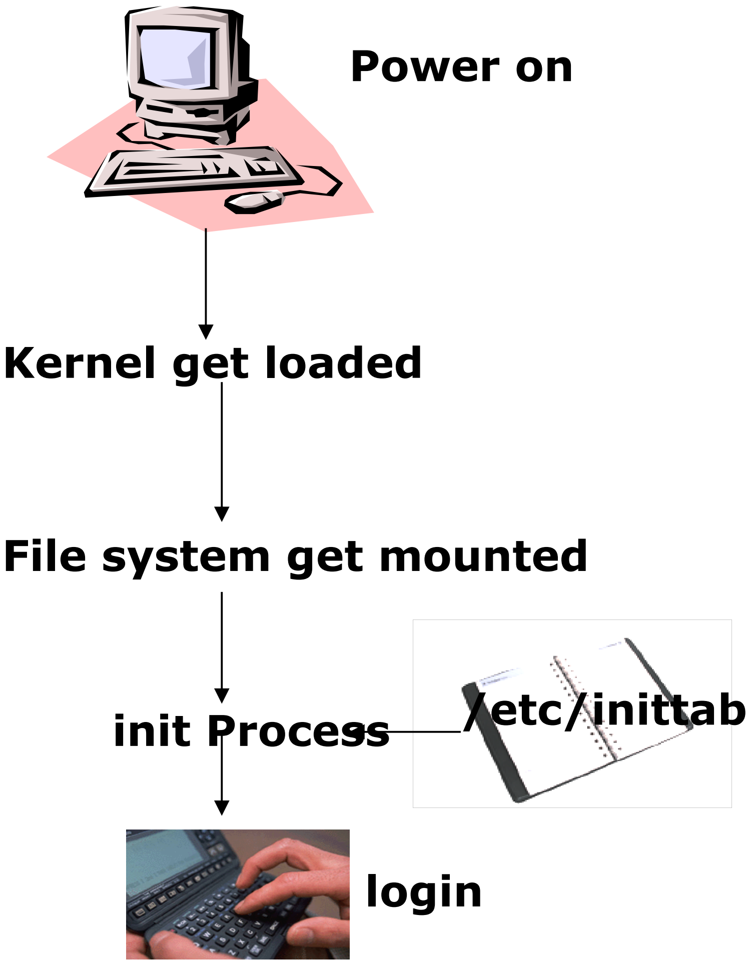
- A Utility program that comes with the UNIX system.

- Features of Shell are:
  - Interactive Processing
  - Background Processing
  - I/O Redirection
  - Pipes
  - Shell Scripts
  - Shell Variables
  - Programming Constructs

Common UNIX Flavors

BSD:	Berkeley, BSD
Solaris:	Sun Microsystems, SVR4/BSD
Ultrix:	Digital Equipment Corporation, BSD
OSF 1:	Digital Equipment Corporation, BSD/SVR4
HPUX:	Hewlett-Packard, SVR4
AIX:	IBM, SVR4 / BSD
IRIX:	Silicon Graphics, SVR4
GNU/Linux:	GNU, BSD/ Posix

System boot up



Power on

Kernel get loaded

File system get mounted

init Process

/etc/inittab

login

Virtual Console

- Though you may not have more than one physical console (a monitor plus a keyboard) connected to your PC, you can use virtual consoles to log in simultaneously to more than one account on your system.

- When any Unix system first boots up, you get a normal Unix login: prompt, so you can log in, and start X11 or do whatever else you would do with a single Unix shell.

- Linux is pretty much the same, except that instead of just one such login:, you get several. These are accessed using the key combinations <ALT><F1>, <ALT><F2>, <ALT><F3> etc.

- Each of the "screens" with you see is known as a "Virtual Console".

Summary

In this module, you have learned about:

- What Is Unix
- History of UNIX
- Characteristics of the UNIX OS
- What is Linux
- Layered Architecture
- Kernel & Shell
- Common UNIX Flavors
- System boot up
- Virtual Console

Session 2 : System Utilities Usages

Objectives

At the end of this module you will learn about:

- Introduction to bash shell
- Getting Started - Shell prompts, Unix Command - arguments & options
- Basic Commands - pwd, date, who, id, whoami, who am i, uname, whereis, tty
- Getting help on commands
- Managing files & directories
- Hard link & Soft Link
- vi Editor
- find command - to search for files and directories
- Filters - tee, wc, tr, cut, sort, head, tail, more, less, grep
- File system commands – df & du
- The awk Programming Language
- Working with run levels
- Shutting the system down
- System Directories

Introduction to bash shell

- bash is a Unix shell written by Brian Fox for the GNU project as a free software replacement for the Bourne shell.

- Released in 1989, it has been distributed widely as the shell for the GNU operating system and as the default shell on Linux, Mac OS X and Darwin.

- As an acronym, it stands for Bourne-again shell, referring to its objective as a free replacement for the Bourne shell.

Getting Started

- Once the system is connected to Unix Server, a user is prompted for a login user id and password.

- The login userid is a unique id on the system. The password is changeable code known only to the user.

- At the login prompt, the user should enter the user id; at the password prompt, the current password should be typed.

- **Note:** Unix is case sensitive. Therefore the login and password should be typed exactly as issued; the login, at least, will normally be in lower case.

The shell prompts

- Once you login successfully, the shell prompt appears at the left side of your screen

- This prompt means the system is waiting for you to type in some unix command.

- **Note:** Generally the shell prompt of $ indicate a regular user login and the prompt of # indicates a root user login.

Unix Command – Arguments & Options

- **Syntax of a Unix Command:**

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

- **See some examples:**

```
ls  -l                                # command with one option
ls -l -a 'or' ls -la         # Command with multiple options

ls  /etc                              # command with one argument
ls  /etc  /dev                 # command with multiple arguments
```

Basic Commands

- **pwd**
  - Displays the current working directory.

- **date**
  - This command displays the current system date and time on the screen.

Basic Commands (Contd.).

- **who**
  - Displays the names of all the users who have currently logged in

- **id**
  - Displays your UID, Primary GID, and Secondary GID’s

- **whoami**
  - Displays the effective user ID

- **who am i**
  - Displays the name of the current user

Basic Commands (Contd.).

- **uname**
  - The uname utility prints information about the current system on the standard output.

- **whereis**
  - Displays the path/ location of a command

- **tty**
  - Prints the terminal’s name

Getting Help on Commands

- The Unix manual, usually called man pages, is available to explain the usage of the Unix system and commands.

- **Syntax:**

  man [options] command_name

- **Common Options**

-k keyword	list command synopsis line for all keyword matches
-M path	path to man pages
-a show	all matching man pages (SVR4)

- info command_name => gives information about commands
- command_name --help =>gives command syntax

touch

- The command **touch** is used to change the time stamp of the file

- **Syntax:**

  touch [options] file

- **Options:**

  -a to change the access time

  -m to change the modification time

  -c no create if the file do not exists

- touch <file> will change the timestamps of the file if the file exists
- If the file does not exist, it will create a file of zero byte size.

Listing Files

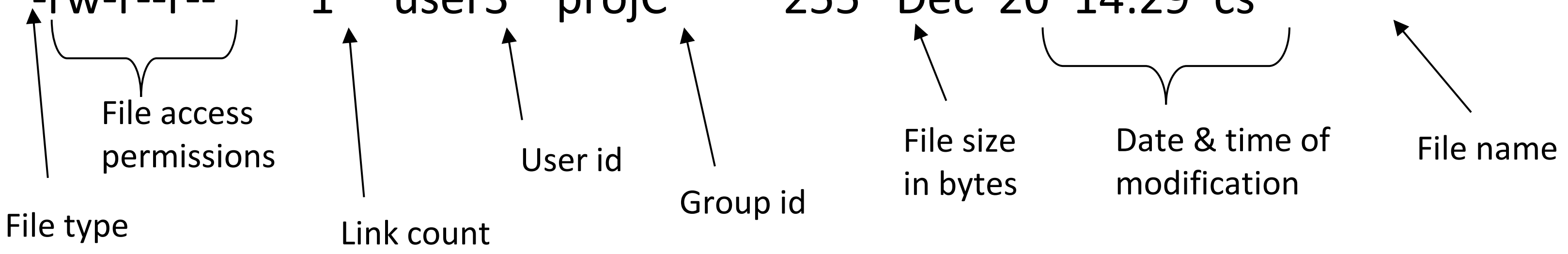
- The command **ls** is used to list the names of files and directories.
- **Syntax:**
  ls [options] [file….]
- **options:**

-l	list in long format
-a	list all files including those beginning with a dot
-i	list inode no of file in first column
-s	reports disk blocks occupied by file
-R	recursively list all sub directories
-F	mark type of each file
-C	display files in columns

Listing the Directory Contents

```
$  ls –l

 total 6
-rwxr-xr-x        1   user1   projA    12373   Dec  15  14:45  a.out
drwxr-xr-x        2   user2   projD     4096   Dec  22  14:00  awkpro
-rw-r--r--    1   user1   projA    12831   Dec  12  13:59  c
-rw-------    1   user1   projA    61440   Dec  15  11:16  core
-rw-r--r--    1   user3   projC      255   Dec  20  14:29  cs
```



File type

File access permissions

Link count

User id

Group id

File size in bytes

Date & time of modification

File name

Meta Characters

Char	Meaning	Example	Possible Output
*	Match with zero or multiple number of any character.	$ ls –l *.c file*	prog1.c, prog2.c, file1 , file2, filebc
?	Match any Single character	$ ls –l file?	filea , fileb, file1
[..]	Match with any single character with in the bracket	$ ls –l file[abc]	filea, fileb,filec
;	Command separator	$cat filea; date	displays the content of filea and displays the current date and time
|	Pipe two commands	$ cat filea | wc -l	Prints the number of lines of filea
()	Group commands, used when the output of the command group has to be redirected	$ (echo “***x.c***”;cat x.c)>out	Redirects the content of x.c with a heading ***x.c*** to the file out

Path names – Absolute & Relative

- The **path** of a file can be represented by either absolute path or relative path.

- **Absolute path:**

  Always begin with the root directory (/). It's a complete roadmap to the file location.

  eg:

  ```
  ls –l   /home/user1/file1
  ```

- **Relative Path:**

  Describes the location of a file/directory with reference to the current directory.

  eg:

  ls –l user1/file1 => assuming that your present working directory is /home, this represents the file /home/user1/file1

Directory Creation

- **The command mkdir create directory**

- **Syntax:**

  mkdir [option] <directory name>

  ```
  $ mkdir  <path>/<directory>
  $mkdir –p <directory1>/<directory2>/<directory3>
  ```

- **Example:**

  $ mkdir project1 =>This creates a directory project1 under current directory

- **Note:** Write and execute permissions are needed for the directory in which user wants to create a directory

Directory Removal

- The command **rmdir** removes directory

- **Syntax**

  rmdir <directory name>

- **Example**

  rmdir project1 => Removes project1 directory in the current directory

  rmdir dir1 dir2 => Remove multiple directories

  rmdir –p dir1/dir2/dir3 => Remove the directory hierarchy

- **Note:** rmdir removes a directory if it is empty and is not the current directory

Changing Directories

- The **cd** command is used to change the directory

cd	=> take to the home directory
cd **..**	=> takes to the parent directory
cd /	=> takes to the root directory

File-Related Commands

File Operation	**Command**
Copying a file	cp
Moving a file	mv
Renaming a file	mv
Removing a file	rm
Displaying a file and concatenating files	cat

Command for copying - cp

- The command **cp** is used to copy files across directories

- **Syntax**

  cp <source file> <new file name>

- **Example**

  cp file1 file2

Command - cp

- **Options to cp**

  -p

  Copies the file and preserves the following attributes

  - owner id
  - group id
  - permissions
  - last modification time

  -r

  - recursive copy; copy subdirectories under the directory if any

  -i

  - interactive; prompts for confirmation before overwriting the target file, if it already exists

Command - mv

- **mv** command is used to move a file, or rename a file

- Preserves the following details
  - owner id
  - group id
  - permissions
  - Last modification time

- **Options**

  -f suppresses all prompting (forces overwriting of target)

  -i prompts before overwriting destination file

Command - rm

- The **rm** command is used to remove a file

- **Syntax :**

  rm file(s)

- **Options**

  -f suppresses all prompting

  -i prompts before deleting destination file

  -r will recursively remove the file from a directory (can be used to delete a directory along with the content)

□ **Caution:** Use “i” option along with “r” to get notified on deletion

Hard Link & Soft Link

- **Linking files**

  - **Hard Link (in the same file system)**

    ```
    $ ln  /usr/bin/clear  /usr/bin/cls
    ```

    - **Note:** Hard links uses the same inode number

  - **Soft Link (possible across file systems - also used to link directories)**

    ```
    $ ln –s  /usr/bin/clear  /home/user1/cls
    ```

vi editor

- **vi** is a visual editor used to create and edit text files.
  - A screen-oriented text editor
  - Included with most UNIX system distributions
  - Command driven
- Categories of commands include
  - Cursor movement
  - Editing commands
  - Search and replace commands
- The vi editor is invoked by the following command:
  - `$ vi filename`

vi modes – the three modes of vi

- **vi** functions in the following modes:
  - Command Mode
  - Insert Mode
  - Last Line Mode

vi command mode

- In the command mode, you can do the following:
  - Issue commands to insert, append, delete, copy etc.
  - Navigate the file
  - Search for text

vi insert mode

- The vi editor switches to insert mode when you issue insert, append or open commands.

- In this mode, you can type text into your file and use the arrow keys to navigate around your file.

- To switch from insert mode to command mode, you need to press the [Esc] key.

vi last line mode

- You can go to the last line mode only from the command mode

- You type **:** in the command mode to switch to the last line mode

- You can do lot of useful actions like saving the file, search and replace commands, issue vi configuration commands.

- You can also issue unix commands from the last line mode

  :! ls

vi configuration commands

- You run the vi configuration commands in the last line mode.

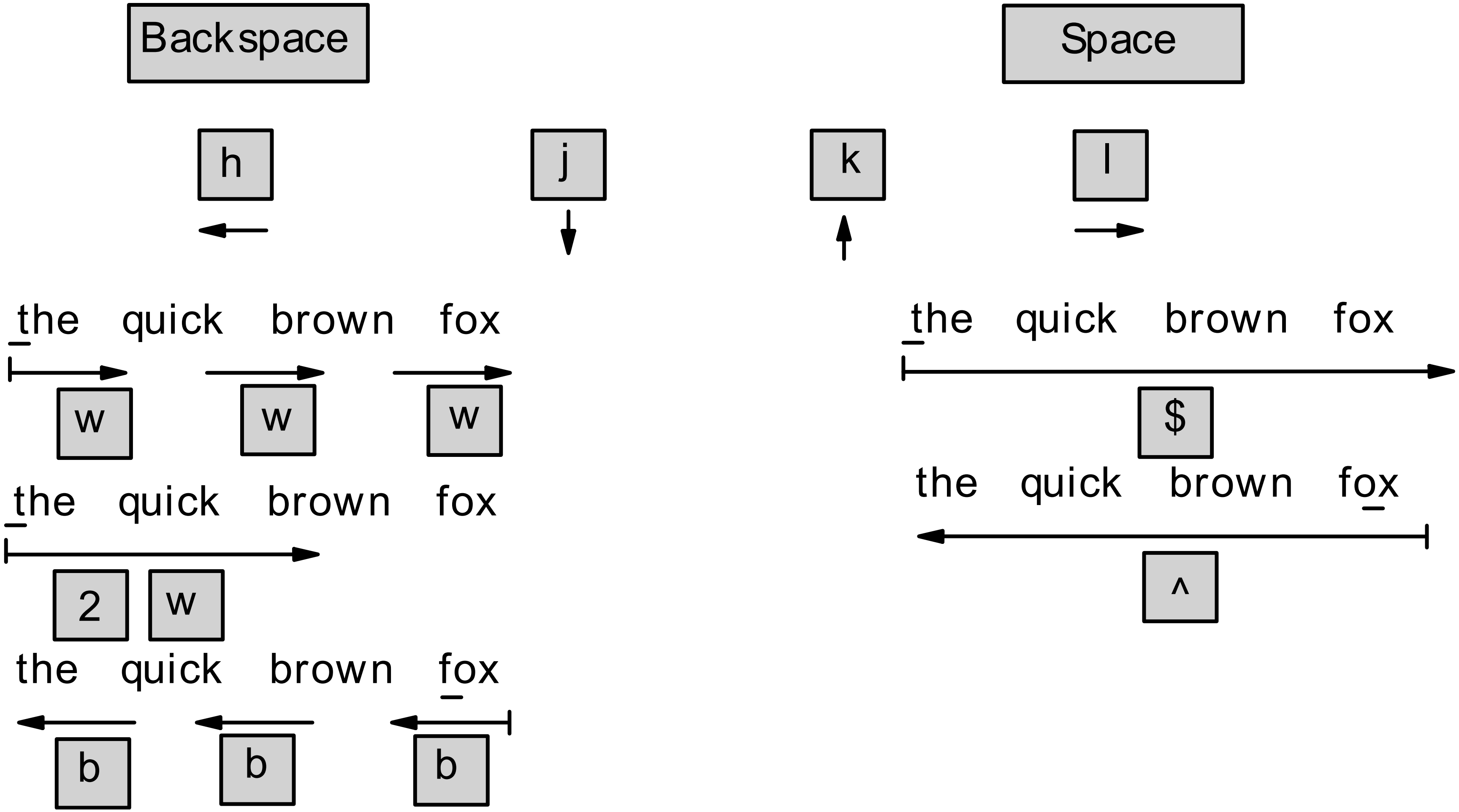
Find below some useful ones:

:se[t] nu => display the line numbers

: se[t] nonu => remove the line numbers

: se[t] showmode => display the current vi mode

Navigation



Backspace

Space

h

j

k

l

the quick brown fox

w w w

the quick brown fox

2 w

the quick brown fox

b b b

the quick brown fox

$

the quick brown fox

^

Editing Commands

- **Text insertion / replacement commands**

i	-	inserts text to the left of the cursor
a	-	inserts text to the right of the cursor
I	-	inserts text at the beginning of the line
A	-	appends text at end of the line
o	-	opens line below
O	-	opens line above
R	-	replaces text from cursor to right
s	-	replaces a single character with any number of characters
S	-	replaces entire line

Editing Commands (Contd.).

- **Commands for deletion**

x	-	to delete character at cursor position
3x	-	to delete 3 characters at cursor position
dw	-	to delete word
2dw	-	to delete 2 word
dd	-	to delete a line
2dd	-	to delete 2 lines
cw	-	to change the text

Editing Commands (Contd.).

- **Yanking (Copying)**

Y	-	copy line into buffer
3Y	-	copy 3 lines into buffer
yw	-	copy a word
p	-	paste buffer below cursor
P	-	paste buffer above cursor
u	-	undo

- **Save and quit**

:w	-	to save
:w!	-	to name a file (:w! filename -> save as)
:x	-	save and quit
:q	-	cancel changes
:q!	-	cancel and quit

Search & Replace Commands

- **The following commands are applicable for vi editor in Linux:**

  /pat searches for the pattern pat and places cursor where pattern occurs.

  / repeat last search

  :%s/old/new/g to change every occurrence in the whole file.

  :#,#s/old/new/g where #,# should be replaced with the numbers of the two lines (say between line no.'s 2 and 5).

- **Example -** :2,5s/am/was/g

find

- Lets user to search set of files and directories based on various criteria

- **Syntax:**

  find [path...] [expression]

  [path]

  - where to search

  [expression]

  - What type of file to search (specified with –type option)
  - What action to be applied (–exec, –print, etc.)
  - Name of the files (specified as part of –name option, enclosed in “ “)

- **Example**

```
find . –name “*.c” -print
```

find (Contd.).

- **Finding files on the basis of file size**

  – size [+ –]n[bc]

  n represents size in bytes (c) or blocks (b) of 512 bytes

- **Examples:**

  find . –size 1000c => lists all files that are exactly
  1000
  bytes in size

  find . –size +1000c => lists all files that are more
  than 1000
  bytes in size

  find . –size –1000c => lists all files that are less than
  1000 bytes in
  size

find (Contd.).

- **Finding files on the basis of access time (atime) or modified time (mtime)**
  - – atime [+-]n
  - – mtime [+-]n

n represents number of days (actually 24 * n hours)

- **Examples:**

find . –atime 2 => lists files accessed exactly 2 days ago

find . –atime +2 => lists files accessed more than 2 days ago

find / –mtime –2 =>lists files modified less than 2 days ago

find (Contd.).

- **Few more options with find:**

-inum n => find files having the inode number n

- type filetype => find files of the file type

f - for ordinary files

d – for directories

-newer fname => find files newer than fname

-perm mode => find files of the specified permissions

-user username => find files owned by the username

find (Contd.).

- **Applying a command on files matching the criteria with – exec and –ok options**

– exec command {} \;

- command is to be applied on the matching files (does not prompt user) – Non interactive.

```
find . -name "*.dat" –exec ls –l {} \;
```

The above command will long list all files with .dat extension in the current and its subdirectories.

-ok command {} \;

- Functionality is similar to –exec, but prompts user before applying the command on the file matching the criteria.

Filter Command – tee

- tee command allows the normal output to the standard output, as well as to a file

- Useful to capture intermediate output of a long command pipeline for further processing, or debugging purpose

- **Examples**
  - who | tee userlist
  - cat - | tee file1 | wc -l

wc

- **wc**

  A filter used to count the number of lines, words, and characters in a disk file or from the standard input

  -l - displays the number of lines

  -w - displays the number of words

  -c - displays the number of characters

tr command

- **tr - translate filter used to translate a given set of characters**

- **Example :**

tr [a-z] [A-Z] < filename => This converts standard input read from lower case to upper case

```
cat lcasefile |  tr “[a-z]” “[A-Z]” >ucasefile
```

tr (Contd.).

- **Useful options for tr**

-s char

Squeeze multiple contiguous occurrences of the character into single char

-d char

Remove the character

Filter Command – cut

- **The cut command is used to extract specified columns/ characters of a text**

Option	remark
-c	used to extract characters
-d	Delimiter for fields
-f	Field no.

- **Examples:**

```
$ cut -c2-5 file1
$ cut -d “|” -f2,3 file1
```

sort

- **Sorts the contents of the given file based on the first char of each line.**

- **Options**

-n	numeric sort (comparison made according to strings numeric value)
-r	reverse sort
-t	specify delimiter for fields
+num	specify sorting field numbers
-knum	specify sorting filed numbers

- **Examples**

$ sort +2 filename => will sort according to the 3rd field.

$ sort –k3 filename => will sort according to the 3rd field.

head

- **The head command will display the top 10 lines by default.**

```
$ head  -3  file1
```

Displays the first 3 lines of the file

tail

- **The tail command displays the last n lines of a file**

```
$ tail  -3 file1           => Displays the last 3 lines of the file
```

Can also specify the line number from which the data has to be displayed till the end of file

```
$ tail  +5  file1
```

Note: The + option is not supported by some unix/linux versions.

Commands – more, less, file

- **more filename** - Displays the file one page at a time

- **less filename** - Displays the file one page at a time

- **file filename** - Display the details of the file type

grep

- **grep – Global Regular Expression Printer is used for searching regular expressions**

- **Syntax**

```
grep <options> <pattern> <filename(s)>
```

- **Related commands**

  egrep & fgrep

grep options

-c displays count of the number of lines where the pattern occurs

-n displays line numbers along with the lines

-v displays all lines except lines matching pattern

-i Ignores case for matching

File system commands

- **df - To check the Filesystem size**
  - **Syntax:**

```
df  [-kmih]

$ df  -k
Filesystem    1k-blocks      Used       Available  Use%  Mounted on
/dev/hda1       8064272    6339628   1314992  83%      /
/dev/hda3               4032161    2016080   1016081   50%      /home
```

- **du - To check file space usage**
  - **Syntax:**

```
du [-ks]
$du –s
83504   .
```

The awk Programming Language

- **awk** is to give Unix a general purpose programming language that handles text (strings) as easily as numbers.

- nawk (new awk) is the new standard for Awk - designed to facilitate large awk programs.

- Looks at data by records and fields

- awk can take the input from
  - files
  - redirection and pipes
  - directly from standard input

- Specify individual records with $0
- Specify individual fields with $1, $2, $3, and so on

awk - examples

- awk -F: ‘{print $1, $3}’ /etc/passwd => Prints the first and 3rd fields(User name and user id) from the file /etc/passwd.

- awk -F: ‘$3>99{print $1, $3}’ /etc/passwd => Prints the first and 3rd fields(User name and user id) from the file /etc/passwd only if the user id is greater than 99.

- awk -F: ‘NR==1,NR==3{print $0}’ /etc/passwd => Prints the first 3 rows from the file /etc/passwd

- who | awk ‘{print $1}’ | uniq => print the unique list of users who are currently logged in.

- awk ‘END{print NR}’ /etc/passwd => print the number of lines in the file /etc/passwd

Run Levels

- Each of the run levels of Linux is a different "operating mode"

  - **single user mode:**

    checking or backing up of file systems done

    only one user – root

  - **multiuser mode:**

    All the file systems are mounted

    System services (daemons) are started

/etc/inittab

- When you boot the system or change run levels with the init or shutdown command, the init daemon starts processes by reading information from the /etc/ inittab file.

- This file defines the following important items for the init process:
  - That the init process will restart
  - What processes to start, monitor, and restart if they terminate
  - What actions to take when the system enters a new run level

Changing the run level

- **who –r or runlevel**

  command to display the current run level of the system

- To switch between run levels you use the init command:

  ```
  #init <Run Level Number>
  ```

- **Examples:**

  init 1 => switch to single user mode

  init 6 => reboot the system

Shutting the System down

- **shutdown usually perform the following activities:**
  - notifies users with “ wall ” command about the system going down
  - send signals to all running processes so that they terminate normally
  - logs users off and kills remaining processes
  - unmounts all the secondary file systems
  - write information about file system status to disk to preserve the integrity of the file system
  - notifies the users to reboot or switch off or moves the system to single user mode

shutdown

Examples

shutdown - system is moved to single user mode

shutdown –g2 - grace period of 2 minutes before shutdown

shutdown –y - don’t prompt the admin “do you want to shutdown”

shutdown –y –g0 - do an immediate shutdown

shutdown –y –g0 –i6 - shutdown and reboot

shutdown –y “system going down for backup”
- customized message
broadcast

System Directories

- **Find below few important system directories**

/	**:**	Root directory.
/bin	:	Command-line executable directory.
/dev	:	Device directory
/etc	:	System configuration files and executable directory.
/lib	:	The library directory.
/home	:	Consists of the users’ home directories
/usr	:	Unix System Resources directory

Summary

In this module, you have learned about:

- Introduction to bash shell
- Getting Started - Shell prompts, Unix Command - arguments & options
- Basic Commands - pwd, date, who, id, whoami, who am i, uname, whereis, tty
- Getting help on commands
- Managing files & directories
- Hard link & Soft Link
- vi Editor
- find command - to search for files and directories
- Filters - tee, wc, tr, cut, sort, head, tail, more, less, grep
- File system commands – df & du
- The awk Programming Language
- Working with run levels
- Shutting the system down
- System Directories

Session 3 : User Management usages

Objectives

At the end of this module you will learn about:

- Understanding File and Directory Permissions
- chown & chgrp
- umask command
- Creating, modifying and deleting User Accounts
- Creating, modifying and deleting Group Accounts
- The su command – switch between users

File Access Permissions

- Refers to the permissions associated with a file with respect to the following:

- **Permission Levels**
  - User (owner) (u)
  - Group (wheel, staff, daemon, etc.) (g)
  - World (guest, anonymous and all other users) (o)

- **Permission Settings**
  - Read (r)
  - Write (w)
  - Execute (x)

File Access Permissions (Contd.).

- No read permission does not allow the user to:
  - List the contents of directory
  - Remove the directory

- No Write permission prevent the user to :
  - Copy files to the directory
  - Remove files from the directory
  - Rename files in the directory
  - Make a subdirectory
  - Remove a subdirectory from the directory
  - Move files to, and from the directory

File Access Permissions (Contd.).

- No execute permission does not allow the user to:
  - display the contents of a directory file from within the directory
  - change to the directory
  - display a file in the directory
  - copy a file to, or from the directory

Changing Permissions - chmod

- **Syntax:**

```
chmod  <category> <operation> <permission> <filename(s)>
                    or
chmod <octal number> filename
```

- **Octal Number**
  - 4 - for read
  - 2 - for write
  - 1 - for execution

- **Examples**

  $ chmod 744 xyz   => This sets read, write and execute permissions for owner, read permission for group and others

  $ chmod u+x file1 => This will add x privilege to the owner.

Command – chown & chgrp

- **chown** – This command changes the owner of the file.

  eg. chown user2 file1 - the new owner of the file file1 is user2.

- **Note: This command can be run by the super user(root) only**

- **chgrp** – This command is used to change the group owner of the file.

  eg. chgrp group2 file1 – this will set the group owner of the file file1 as group2.

- **Note: This command can be run by a normal user to change the group owner to any of the groups to which he belongs to. If the group has to be changed to some other group, to which he do not belongs to, only superuser can change it.**

umask command

- umask value is used to set the default permission of a file and directory while creating
- **umask** command (without any arguments) is used to see the default mask for the file permission
- Default umask value will be set in the system environment file like/ etc/ profile
- The command **umask 022** will set a mask of 022 for the current session
  - If you create a new file now, the file permission will be 644
  - And the directory permission will be 755

User Management

- Users refer to either people which means accounts tied to physical users or accounts which exist for specific applications to use.

- Groups are used to club users together for a common purpose. Users within a group can generally read, write, or execute files owned by that group.

- Each user and group has a unique numerical identification number called a userid (UID) and a groupid(GID), respectively.

- A user who creates a file is also the owner and group owner of that file.

User Management (Contd.).

- System Administrator uses commands or GUI tool to create User credentials (ID and password)
- **Commands to create a user**
  useradd or adduser
- **Commands to create a group**
  groupadd or addgroup
- **Commands to modify a user & group**
  usermod groupmod
- **Commands to delete a user & group**
  userdel groupdel
- **Command to set/reset password**
  passwd <username>

Creating user account – useradd command

- **Syntax:**

  useradd [-c comment] [-d home_dir] [-e expire_date] [-f inactive_time] [-g initial_group] [-G group[,...]] [-m] [-k skeleton_dir] [-p passwd] [-s shell] [-u uid [-o] login

- **Examples**

  # useradd User1 => command to create a locked user account User1

  # passwd User1 => Sets the password for the new user

- **Note:**The above commands make entries in the associated file /etc/passwd & /etc/shadow.

  #useradd –D => will display the default values used by the useradd command.

Modifying use account – usermod command

- **Syntax:**

  usermod [-c comment] [-d home_dir [-m]] [-e expire_date] [-f inactive_time] [-g initial_group] [-G group[,...]] [-l login_name] [-p passwd] [-s shell] [-u uid [-o]] [-L|-U] login

- **Note: Most** of the options associated with the useradd command are available with the usermod command as well.

Deleting user accounts – userdel

- **Syntax:**

```
userdel  [-r]  login
```

-r Files in the user's home directory will be removed along with the home directory itself and the user's mail spool.

- **Note: Files** located in other file systems will have to be searched for and removed manually.

Without the option – r, the command userdel only deletes the user. His home directory will remain intact.

Adding Group Account – groupadd command

- **Syntax:**

  groupadd [-g gid [-o]] group

  -g gid => specifies the Group ID of the group

groupadd Group1 => create a new group Group1

Modifying Group Accounts – groupmod command

- **Syntax**

```
groupmod  [-g gid  [-o]] [-n group_name ] group
```

-n group_name => specifies the new group name.

This option is used for renaming a Group Account

Deleting Group Account - groupdel

- **Syntax**

```
groupdel groupname
```

- **Example:**

```
# groupdel  Group1
```

User Management – GUI Tool

- In RHEL5 , you can use the GUI Tool User Manager to manage the users & groups in a GUI environment.

- To use the User Manager, you must be running the X Window System, have root privileges, and have the system-config-users RPM package installed.

- To start the User Manager from the desktop,
  - go to System (on the panel) -> Administration -> Users & Groups.
  - you can also type the command system-config-users at a shell prompt (for example, in an XTerm or a GNOME terminal).

passwd

- **To change a user password**

```
$  passwd
Changing password for username
New UNIX password:
Reenter UNIX password:
```

User Related Files

- **/etc/passwd** This file stores all the user information except the password
- **/etc/shadow** This file stores the encrypted passwords and all password related information
- **/etc/group** This file stores all the groups in the system along with the secondary group members
- **/etc/default/useradd** This files stores the default values used by useradd program
- **/etc/login.defs** This file defines the site specific configuration for the shadow password suite.

/etc/passwd

- **Have a look at the example given below of /etc/passwd file from a Unix system**

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
```

Find below the columns of the file:

LoginID - user's login name

x - represents a placeholder for the user's encrypted password

UID - user ID number used by the system to identify the user

GID - GID number which identifies the user's primary group

comment - comment about the user – normally a user's full name

home_directory - specifies the full path name to the user's home directory

login_shell - defines the user's login shell

/etc/shadow

- **Find below the first two lines of an /etc/shadow file from a linux system:**

```
root:5RiJS.yvdGBkU:13255:0:99999:7:::
bin:*:13255:0:99999:7:::
```

Fields in the /etc/shadow file

loginID - user’s login name

encrypted_passwd - encrypted password

lastchg - the number of days between Jan 1, 1970 and the last password modification date

min - minimum no. of days between password changes

max - maximum password age (number of days the password is valid)

warn - The 'warn' field flashes the information about the number of days before the password expires

inactive - number of inactive days allowed for the user before the account getting locked

expire - The date when the user account expires

reserved field - For future use

/etc/group

- **Find below the first two lines of an /etc/group file from a linux system**

  root:x:0:root

  bin:x:1:root,bin,daemon

- **Fields in the /etc/group file**

goupname - name assigned to the group

group-passwd - allows a non group member to work as group member on supply of this password

gid - group id

userlist - list of user names for whom this group is their secondary group.

/etc/default/useradd

- **Find below content of /etc/default/useradd file from a linux system**

```
 # useradd defaults file
 GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
 CREATE_MAIL_SPOOL=yes
```

/etc/login.defs

- **Find below few lines of /etc/login.defs file   from a linux system**

```
 MAIL_DIR                              /var/spool/mail
 PASS_MAX_DAYS                   99999
 PASS_MIN_DAYS                   0
 PASS_MIN_LEN                    5
 PASS_WARN_AGE             7
UID_MIN                                500
GID_MIN                                500
 CREATE_HOME                     yes
```

su command

- The su command is used to switch the user to another user

- System administrator should not login as “ root ” . They should login as a normal user and run su command to switch user

- **Examples**

su => Profile will not change

su – => Profile will change to the root’s

su - <username> => Switch to a user with profile change

Summary

In this module, you have learned about:

- Understanding File and Directory Permissions
- chown & chgrp
- umask command
- Creating, modifying and deleting User Accounts
- Creating, modifying and deleting Group Accounts
- The su command – switch between users

Session 4 : Process Management & IPC

Objectives

At the end of this module you will learn about:

- Discussion about pipes and sockets
- Process, Process Status, Foreground and Background Processes
- Killing Processes
- Process Priority
- Accessing system and boot security logs

Pipes and Sockets

- Pipes and Sockets are special files that programs use to communicate with one another.

- ## Unnamed pipe:

  A simple example of using an unnamed pipe is the command:

  ls | grep x => bash and other shells run both commands, connecting the output of the first to the input of the second.

- ## Named pipe:

  Instead of a conventional, unnamed, shell pipeline, a named pipeline makes use of the file system. It is explicitly created using mkfifo() or mknod() and two separate processes can access the pipe by name — one process can open it as a reader, and the other as a writer.

- **Example: Execute in Virtual Console1** **Execute in Virtual Console2**

  #mkfifo pipe1 ; ls –l > pipe1 # cat < pipe1

  The output of the first command shows up on the second console.

Named pipe - Examples

- **Simple Example:**

Create named pipes

```
$ mkfifo myfifo ;  mkfifo myfifo2
```

Start a process and redirect standard output stream to the pipe

```
$ ls -l > myfifo &
[1] 9823
```

start a process that reads contents from the pipe

```
$ cat myfifo
insgesamt 0
prw-r--r-- 1 mario users 0 13. Nov 22:37 myfifo
prw-r--r-- 1 mario users 0 13. Nov 22:37 myfifo2
```

Note : You can create a dump of an Oracle database schema and imports the dump into another remote Oracle database. The size of the database dump does not matter, because the data stream is not stored to hard disks. The data stream will be redirected to the Oracle import tool.

Unix Domain Socket or IPC Socket

- A Unix domain socket or IPC socket is a data communications endpoint for exchanging data between processes executing within the same host operating system.

- While similar in functionality to named pipes Unix domain sockets may be created as byte streams or as datagram sequences, while pipes are byte streams only.

- The programmer's application interface (API) for Unix domain sockets is similar to that of an Internet socket, but does not use an underlying network protocol for communication.

What is a Process

- 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.

ps - process status

- 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.

- **Options**

-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
-l	list the information like process state as well

Foreground & Background Processes

Foreground Processes

- By default the processes get executed in the foreground.
- Only one process can get executed in the foreground at a time in one terminal.

Background Processes

- If the command terminates with an ampersand (&), UNIX executes the command in the background
- Background processes enables the user to do more than one task at a time.
- Shell returns by displaying the process ID (PID) and job id of the process

Controlling Background Process

- **jobs**

  List the background process

- **fg %<job id>**

  Runs a process in the foreground

- **jobs –l**

The kill Command

- **kill** - Kills or terminates a process

- **kill command send a signal to the process**
  - The default signal is 15 (SIGTERM)

- **kill -9 (SIGKILL)**
  - Terminates the process abruptly

- **pkill : command used to kill a process by name**

  ```
  pkill  <command name>
  pkill  -9  <command name>
  ```

nohup Command

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

- **Syntax**

```
nohup command arguments
```

- **Example:**

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

Process priority

- **nice**
  - This command is used to alter the priority of jobs

- **renice**
  - Used to change the priority of a running process

Accessing Logs

- Logs serve several purposes:
  - They help us troubleshoot virtually all kinds of system and application problems.
  - They provide valuable early warning signs of system abuse.
  - When all else fails (whether that means a system crash or a system compromise), logs provide us with crucial forensic data.

Configuring syslog

- The **syslog.conf** file is the main configuration file for the **syslogd** which logs system messages on unix systems. This file specifies rules for logging.

- Every rule consists of two fields, a selector field and an action field. These two fields are separated by one or more spaces or tabs.

- The selector field specifies a pattern of facilities and priorities belonging to the specified action.

- Lines starting with a hash mark ("#") and empty lines are ignored.

Configuring syslog (Contd.).

- **Some sample syslog entries:**

  *.=crit;kern.none /var/adm/critical => This will store all messages with the priority crit in the file /var/adm/critical, except for any kernel messages.

  kern.crit @Server1 => directs all kernel messages of the priority crit and higher to the remote host Server1.

  kern.* /var/adm/kernel => direct any message that has the kernel facility to the file /var/adm/kernel

  kern.crit /dev/console => directs these messages to the actual console, so the person who works on the machine will get them, too.

- **Note:** After making any kind of changes in the syslog.conf file, you need to restart the syslogd daemon or send the HUP signal to the syslog daemon to get the changes effective.

Important log files

- Most of the log files are stored at the location / var/ log

- Common Linux log files name and usage

/var/log/messages	: General message and system related stuff.
/var/log/auth.log	:Authentication logs.
/var/log/kern.log	: Kernel logs.
/var/log/boot.log	: System boot log.
/var/log/secure	: Authentication log.
/var/log/utmp	: Online users data – used by **who** command
/var/log/wtmp	: Login details – used by **last** command

dmesg is a command on most Linux and Unix based operating systems that prints the message buffer of the kernel.

Summary

In this module, you have learned about:

- Discussion about pipes and sockets
- Process, Process Status, Foreground and Background Processes
- Killing Processes
- Process Priority
- Accessing system and boot security logs

Session 5 : Unix Back up usges

Objectives

At the end of this module you will learn about:

- Backup through tar /cpio / dd commands
- Recovery single/multiple files

Backup & Restore

- One of the major activities to enable the availability of system and software.

- Each Unix flavor got its own backup functions.

- General backup facility will allow to use the backed up files across flavors.

- Commercial backup facilities are also available.

Type of Backups

- Full Backup
  - Take the backup of all files
- Incremental Backup
  - Take the backup of all files modified after the last full backup or incremental backup
- Differential Backup
  - Take the backup of all files modifies after the last full backup

Backup Utilities: Tape Archive - tar

- **tar** is an archiving utility to store and retrieve files from an archive, known as tarfile.

- Though archives are created on a tape, it is common to have them as disk files as well.

- **Syntax**

  tar c|t|x [vf destination] source...

- **Examples**

```
$ tar -cf  tar1  emp          => take a backup of emp directory into
                                           the tarfile tar1
$ tar –tf  tar1               => List the files & directories in the tar
                                           file.
```

Tape Archive - tar

- **Examples:**

  Create a new tar file containing all .dat files (assuming a.dat, b.dat and c.dat exist)

  ```
  $ tar –cf  mytar  *.dat
  ```

  ```
  $ tar –xf mytar
  ```

  => Restores the data from the backup.

Backup through cpio

- The cpio command is one of the most commonly used Linux back up tools.

- The cpio command has two unusual features. Unlike tar, in which the files to back up are typed in as part of the command, cpio reads the files to work with from the standard input (in other words, the screen).

- This feature means that cpio must be used as part of a multiple command or with a redirection pipe.

- cpio must always be used with one of three flags – extract , create and pass-through.

Backup & restore using cpio - Examples

- **Example: To take the backup of the directory /home/User1**

```
# find  /home/User1  -print | cpio  -ocv  > /opt/User1_backup.cpio
```

- **Command to restore the directory from the backup:**

```
# cd  /home/User1
# cpio –icuvd   < /opt/User1_backup.cpio
```

- **Copy a directory structure – using the pass through mode.**

```
#find  . –depth | cpio –pmdv /export/out
```

Backup & restore using dd command

- The dd command is used by the Linux kernel Makefiles to make boot images.

- Like most well-behaved commands, dd reads from its standard input and writes to its standard output, which could be altered by a command linen specification.

- This allows dd to be used in pipes, and remotely with the rsh remote shell command.

Backup & restore using dd command - Examples

Full Hard Disk Copy:

```
# dd  if=/dev/hdb  of=/dev/hdc
# dd  if=/dev/hdb  of=/opt/hdb_backup
# dd  if=/dev/hdb | gzip > /opt/hdb_backup.gz
```

Restore Backup of hard disk copy

```
# dd  if=/opt/hdb_backup  of=/dev/hdb
# gzip -dc /opt/hdb_backup.gz | dd of=/dev/hdb
```

MBR backup

In order to backup only the first few bytes containing the MBR and the partition table you can use dd as well.

```
#dd  if=/dev/hda  of=/opt/mbr_backup count=1 bs=512
```

MBR restore

```
dd  if=/opt/mbr_backup  of=/dev/hda
```

- **Note:** Add "count=1 bs=446" to exclude the partition table from being written to disk. You can manually restore the table.

Backup & Restore

- One of the major activities to enable the availability of system and software.
- Each Unix flavor got its own backup functions.
- General backup facility will allow to use the backed up files across flavors.
- Commercial backup facilities are also available.

Type of Backups

- Full Backup
  - Take the backup of all files

- Incremental Backup
  - Take the backup of all files modified after the last full backup or incremental backup

- Differential Backup
  - Take the backup of all files modifies after the last full backup

Backup Utilities: Tape Archive - tar

- **tar** is an archiving utility to store and retrieve files from an archive, known as tarfile.

- Though archives are created on a tape, it is common to have them as disk files as well.

- **Syntax**

  ```
  tar c|t|x [vf destination] source...
  ```

- **Examples**

  ```
  $ tar -cf  tar1  emp          => take a backup of emp directory into
                                            the tarfile tar1
  $ tar –tf  tar1               => List the files & directories in the tar
                                            file.
  ```

Tape Archive - tar

- **Examples:**

  Create a new tar file containing all .dat files (assuming a.dat, b.dat and c.dat exist)

  $ tar –cf mytar *.dat

  $ tar –xf mytar   =>   Restores   the   data   from the

  backup.

Backup through cpio

- The cpio command is one of the most commonly used Linux back up tools.

- The cpio command has two unusual features. Unlike tar, in which the files to back up are typed in as part of the command, cpio reads the files to work with from the standard input (in other words, the screen).

- This feature means that cpio must be used as part of a multiple command or with a redirection pipe.

- cpio must always be used with one of three flags – extract , create and pass-through.

Backup & restore using cpio - Examples

- **Example: To take the backup of the directory /home/User1**

```
# find  /home/User1  -print | cpio  -ocv  > /opt/User1_backup.cpio
```

- **Command to restore the directory from the backup:**

```
# cd  /home/User1
# cpio –icuvd   < /opt/User1_backup.cpio
```

- **Copy a directory structure – using the pass through mode.**

```
#find  . –depth | cpio –pmdv /export/out
```

Backup & restore using dd command

- The dd command is used by the Linux kernel Makefiles to make boot images.

- Like most well-behaved commands, dd reads from its standard input and writes to its standard output, which could be altered by a command linen specification.

- This allows dd to be used in pipes, and remotely with the rsh remote shell command.

Backup & restore using dd command - Examples

Full Hard Disk Copy:

```
# dd  if=/dev/hdb  of=/dev/hdc
# dd  if=/dev/hdb  of=/opt/hdb_backup
# dd  if=/dev/hdb | gzip > /opt/hdb_backup.gz
```

Restore Backup of hard disk copy

```
# dd  if=/opt/hdb_backup  of=/dev/hdb
# gzip -dc /opt/hdb_backup.gz | dd of=/dev/hdb
```

MBR backup

In order to backup only the first few bytes containing the MBR and the partition table you can use dd as well.

```
#dd  if=/dev/hda  of=/opt/mbr_backup count=1 bs=512
```

MBR restore

```
dd  if=/opt/mbr_backup  of=/dev/hda
```

- **Note:** Add "count=1 bs=446" to exclude the partition table from being written to disk. You can manually restore the table.

Session 6: Unix Shell Script

Objectives

At the end of this module you will learn about:

- Unix Shell
- Configuration Scripts
- Shell Variables
- Environment Variables
- The cat command
- Standard Files
- I/O Redirection
- Sample Shell script
- Executing a Shell script
- Passing parameters to Shell script
- Doing arithmetic & Comparison operations
- Condition checking, Iterations, case statement & Functions
- Debugging shell scripts

Unix Shell

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

Additional Shell Features

- In addition to the basic features, other additional shell features are listed below:
  - Maintaining command history (C, korn and bash)
  - Renaming (aliasing) a command (C, korn, bash)
  - Command editing (C, korn and bash)
  - Programming language (all shells)

Configuration Scripts

	sh	ksh	csh	bash
System Profile	/etc/profile	/etc/profile	/etc/login	/etc/profile
User profile	~/.profile	~/.profile	~/.login	~/.bash_profile
Script file		~/.kshrc	~/.cshrc	~/.bashrc

~ is used to represent the home directory of the user

Scripting

- 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:
    - **Local variables:** Local variables are only available in the current shell. Using the set built-in command without any options will display a list of all variables
    - **Environment variables or global variables:** available in all shells – inherited to the sub processes. The env or printenv commands can be used to display global variables.

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

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 (Contd.).

➢ 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	=> search path for the binaries
BASH_ENV	=> bashrc path
HOME	=> home directory
PWD	=> working directory
SHELL	=> login shell
TERM	=> Terminal Type
PS1	=> Primary Prompt
PS2	=> Secondary Prompt
MAIL	=> path of the mail box
USER	=> user name
LOGNAME	=> user name

cat

- The cat command takes its input from the keyboard, and sends the output to the monitor.
- We can redirect the input and output using the redirection operators.
- **Examples**

```
$ cat  > file1
Type the content here
press <ctrl d>
$ cat file1
Displays the content of the file
$cat  >>  file1
```

This will append standard input to the content of file1.

Standard Files

- Standard Input file
  - Keyboard, file descriptor is 0
- Standard Output file
  - Monitor, file descriptor is 1
- Standard Error file
  - Monitor, file descriptor is 2

I/O Redirection

Redirection Operators

< file	redirect standard input from file
> file	redirect standard output to file
2> file	This will redirect standard error to file
2>&1	merge standard error with standard output

Examples

```
$ cat > abc

$ ls –l > outfile

$ cat xyz abc > outfile 2> errfile

$ cat xyz abc > outfile 2>&1
```

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  “Type your name here”
read name
echo “Welcome $name”
echo “Hello $USER…....”
echo “Your Home Directory is $HOME”
echo “Your Shell is $SHELL”
# -------------------------------------------------------------
```

Executing Shell Script

- **There are two ways of executing a shell script as a sub process:**
  - By passing the name of the shell script as an argument to the shell. For example:

    ```
    $ bash sample_script.sh
    ```

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

    ```
    $ ./sample_script.sh
    ```

- **To execute the shell script in the current shell:**
  - In the above cases, the specified shell will start as a subshell of your current shell and execute the script. To execute the script in the current shell, you source it as below. The script don't need execute permission in this case.

    ```
    $ source script1.sh
    ```

Passing Parameters to Scripts

- parameter can be passed to a shell script .

- The command line 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 Bourne shell(sh). In case of more than 9 parameters, remaining parameters can be accessed by using the shift command.

- The bash shell do not have any such limitations on the number of parameters($1-$9).Shift command is supported by bash as well.

Built-in variables

- **Following are built-in variables supported**
  - $1,$2… - positional arguments
  - $* - all arguments
  - $@ - all arguments
  - $? - exit status of previous command executed
  - $$ - PID of the current process
  - $! - PID of the last background process
  - $0 - Expands to the name of shell or shell script
  - $# - Expands to the number of positional parameters

Passing Parameters to Scripts

- **Consider following shell script:**

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

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

Passing Parameters to Scripts (Contd.).

- The shell parameters are passed as strings.

- To pass a string containing multiple words as a single parameter, it must be enclosed within quotes.

- **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`
```

- **Note:** There should be no space around the assignment operator whereas there should be a space around the arithmetic operator in the expr command. Also note the presence of the command substitution operator.

Arithmetic Expansion in bash shell

- Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

  $((EXPRESSION))

  $ echo $((10*20))   # The result 200 is displayed on   the screen

- All tokens in the expression undergo parameter expansion, command substitution, and quote removal. Arithmetic substitutions may be nested.

- The operators are roughly the same as in the C programming language.

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 (Contd.).

- **Numerical Comparison Operators**

-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 (Contd.).

- **General syntax**

```
test expression
        or
 [ expression ]

test integer1 operator integer2
                   OR
[ integer1 operator integer2 ]
```

Using the test Command (Contd.).

- **String comparison Operators**

  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 (Contd.).

- **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 (Contd.).

- **File comparison operators**

  -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 (Contd.).

- **File Comparison operators**

  -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 (Contd.).

- **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 pesent
                        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
```

Using for loop

All the shells provides for loop.

Syntax:

for variable in list ; do COMMANDS ; done

- **Example:**

```
for i in 1 3 5 7 9
        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 command substitution for specifying LIST items

- **Find below an example demonstrating the use of a for loop that makes a backup copy of the .cc files in the current directory – uses command substitution for specifying the LIST items.**

```
# ls  *.cc > list                # create a file with the names of the .cc files
```

Shell script to take the backup:

```
#!  /bin/bash
for i in `cat list`
do
            cp  "$i"  "$i".bak
done
```

Using the content of a variable to specify LIST items

- **You can use a variable to specify the LIST items**
- **Example** to check the existence of a set of files and displaying message:

```
#!/bin/bash
files=“file1
file2
file3”
for  file  in  $files ;  do
          if  [ ! -e "$file" ]
           then
          echo "$file does not exist.";
          fi
done
```

Three-expression bash for loops syntax

- **This type of for loop share a common heritage with the C programming language. It is characterized by a three-parameter loop control expression; consisting of an initializer (EXP1), a loop-test or condition (EXP2), and a counting expression (EXP3).**

```
for(( EXP1; EXP2; EXP3 ))
 do
        command1
        command2
done
```

- **A representative three-expression example in bash as follows:**

```
#!/bin/bash
for(( a=1; a<=5; a++ )) ;do; echo "Welcome $a times...“; done
```

Using range of values in the list

- **The bash version 3.0+ has inbuilt support for setting up ranges:**

```
#! /bin/bash
for i in {1..10} ;          do  ;  echo  “ Value of i:  $i "  ; done
```

- **bash v4.0+ has inbuilt support for setting up a step value** using {START..END..INCREMENT} syntax:

```
#!/bin/bash
echo "Bash version ${BASH_VERSION}
        for i in {0..10..2} ; do ; echo "Welcome $i times“;
                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**

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

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

until loop

- The until loop is very similar to the while loop, except that the loop executes until the TEST-COMMAND executes successfully.

- As long as this command fails, the loop continues. The syntax is the same as for the while loop:

```
until TEST-COMMAND
do
        CONSEQUENT-COMMANDS
done
```

until loop example

- **Example – script to print all the command line arguments:**

```
#! /bin/bash
count=1
until  [ "$*" = "" ]
do
  echo "Argument number  $count :  $1 "
  shift
  count=`expr $count + 1`
done
```

break and continue

- ## Conditional exit with break

You can do early exit with break statement inside the for loop. You can exit from within a FOR, WHILE or UNTIL loop using break

```
if condition
then
        break
end if
```

- ## Early continuation with continue statement

To resume the next iteration of the enclosing FOR, WHILE or UNTIL loop use continue statement.

The case Statement

- **The structure of case statement**

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

Example

- **Program to add and subtract 2 numbers using case**

```
  #!/bin/bash
  echo “enter 2 nos “ ; read num1 ; read num2
  echo “enter 1 (for addition) or 2 (for subtraction)”
  read choice
  case $choice in
          1)  res=`expr $num1 + $num2`
              echo result is $res;;
          2)  res=`expr $num1 -  $num2`
              echo result is $res;;
          *) echo invalid input;;
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 (Contd.).

- 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 (Contd.).

- **Another example**

```
#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

➢ Options to 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. Comments will not be visible in the output.

  $ bash –x script-file

- **“-f” option:**

  disable file name generation using metacharacters (globbing)

Summary

In this module, you have learned about:

- Unix Shell
- Configuration Scripts
- Shell Variables
- Environment Variables
- The cat command
- Standard Files
- I/O Redirection
- Sample Shell script
- Executing a Shell script
- Passing parameters to Shell script
- Doing arithmetic & Comparison operations
- Condition checking, Iterations, case statement & Functions
- Debugging shell scripts

References

- Das, Sumitabha. *Unix : Concepts and Applications*. New Delhi: Tata McGraw-Hill, 2008.

- Machtelt Garrels (2008). *Bash Guide for Beginners.* Retrieved on October 6, 2011, from, http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

- Paul K. Andersen. Just Enough UNIX. New York: McGraw-Hill, Inc, 2006.

- Cyberciti.biz (2011). Bash C Style For Loop Example and Syntax. Retrieved on May 21, 2012, from, http://www.cyberciti.biz/faq/linux-unix-applesox-bsd-bash-cstyle-for-loop

- Cyberciti.biz (2012). HowTo: Iterate Bash For Loop Variable Range Under Unix / Linux. Retrieved on May 21, 2012, from, http://www.cyberciti.biz/faq/linux-unix-applesox-bsd-bash-cstyle-for-loop