UNIX

00011110 00011110 00011110 00011110 00011110 00011110 00011110 00011110

is user-friendly.

It's just very selective about who its friends are.

Operating System

- An “Operating System” (OS) is the program which starts up when you turn on your computer and runs underneath all other programs - without it nothing would happen at all.
- An operating system is a manager. It manages all the available resources on a computer, from the CPU, to memory, to hard disk accesses.
- Tasks the operating system must perform:
    - Control Hardware
    - Run Applications
    - Manage Data and Files
    - Security

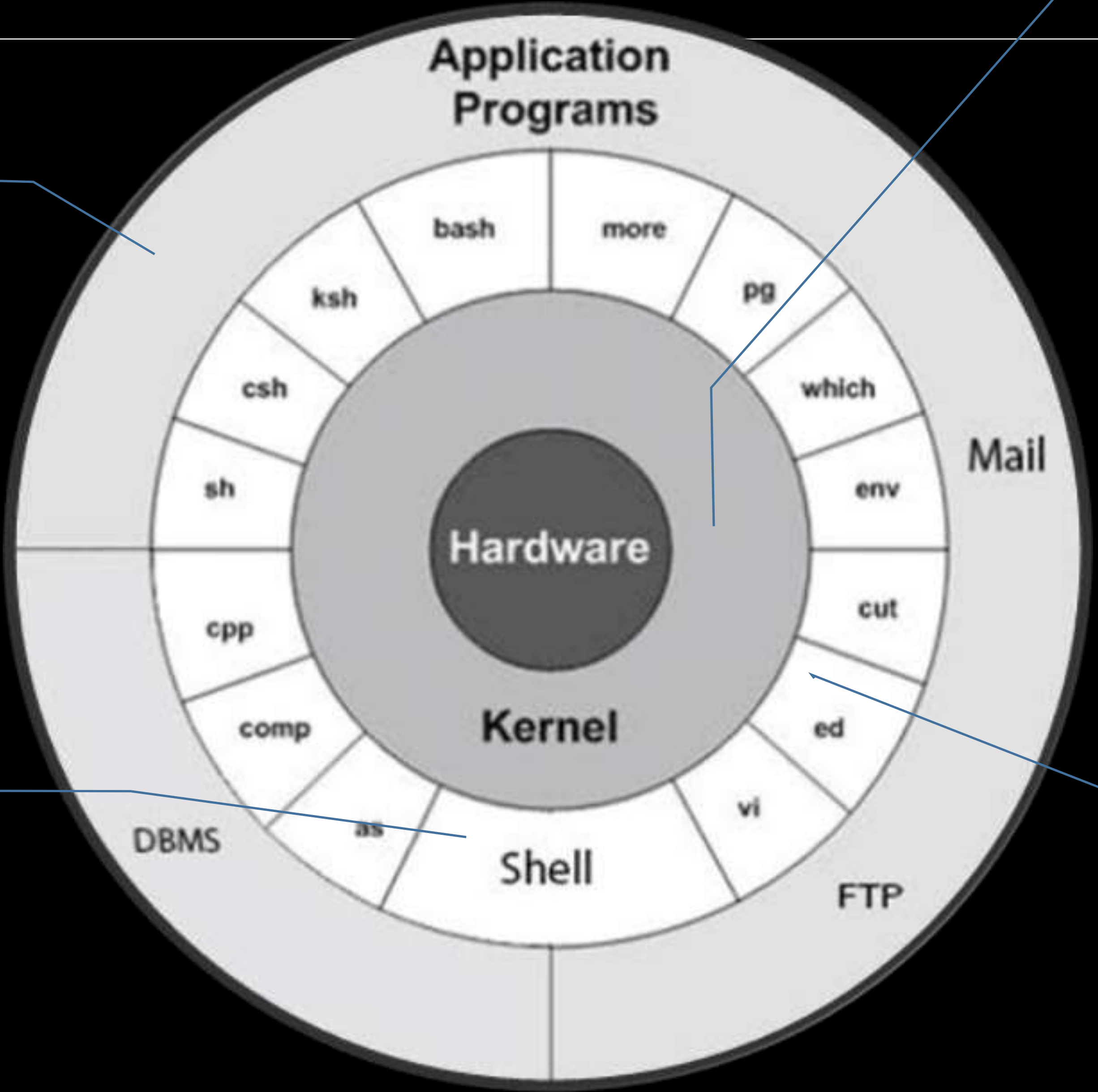
What is Unix?

- A multi-user networked operating system
  - “Operating System”
    - Handles files, running other programs, input/output
    - Looks like DOS…but more powerful
    - The internet was designed on it, thus networking is an intrinsic part of the system
  - “Multi-user”
    - Every user has different settings and permissions
    - Multiple users can be logged in simultaneously
- Tons of fun!!! ☺

Unix

- Age
  - Born in 1970
- Where
  - AT&T Bell Labs
- Founders
  - Ken Thompson and Dennis Ritchie.
- Sun, IBM, HP are the 3 largest vendors of Unix
  - These Unix flavors all run on custom hardware

Unix Architecture



Heart of the Operating System

Files, Directories, Databases

Processes requests C Shell, Bourne Shell, Korn Shell

Command & Utilities

General Characteristics

- Multi-user & Multi-tasking –
    - most versions of UNIX are capable of allowing multiple users to log onto the system, and have each run multiple tasks. This is standard for most modern OSs.

General Characteristics

- Over 40 Years Old –
    - UNIX is over 40 years old and it's popularity and use is still high
    - Over these years, many variations have spawned off and many have died off, but most modern UNIX systems can be traced back to the original versions. It has endured the test of time.
    - For reference, Windows at best is half as old (Windows 1.0 was released in the mid 80s, but it was not stable or very complete until the 3.x family, which was released in the early 90s).

General Characteristics

- Large Number of Applications –
    - there are an enormous amount of applications available for UNIX operating systems
    - They range from commercial applications such as CAD, Maya, WordPerfect, to many free applications

General Characteristics

- Free Applications and Even a Free Operating System –
  - Of all of the applications available under UNIX, many of them are free
  - The compilers and interpreters used in most of programming run on UNIX

General Characteristics

- Less Resource Intensive –
    - In general, most UNIX installations tend to be much less demanding on system resources
    - In many cases, the old family computer that can barely run Windows is more than sufficient to run the latest version of Linux.

General Characteristics

- Internet Development –
  - Much of the backbone of the Internet is run by UNIX servers
  - Many of the more general web servers run UNIX with the Apache web server - another free application
-

Flavors of UNIX

- **AIX** - developed by IBM for use on its mainframe computers
- **BSD/OS** - developed by Wind River for Intel processors
- **HP-UX** - developed by Hewlett-Packard for its HP 9000 series of business servers
- **IRIX** - developed by SGI for applications that use 3-D visualization and virtual reality
- **QNX** - a real time operating system developed by QNX Software Systems primarily for use in embedded systems
- **Solaris** - developed by Sun Microsystems for the SPARC platform and the most widely used proprietary flavor for web servers
- **Tru64** - developed by Compaq for the Alpha processor

UNIX interfaces

Graphical User Interfaces (GUIs)

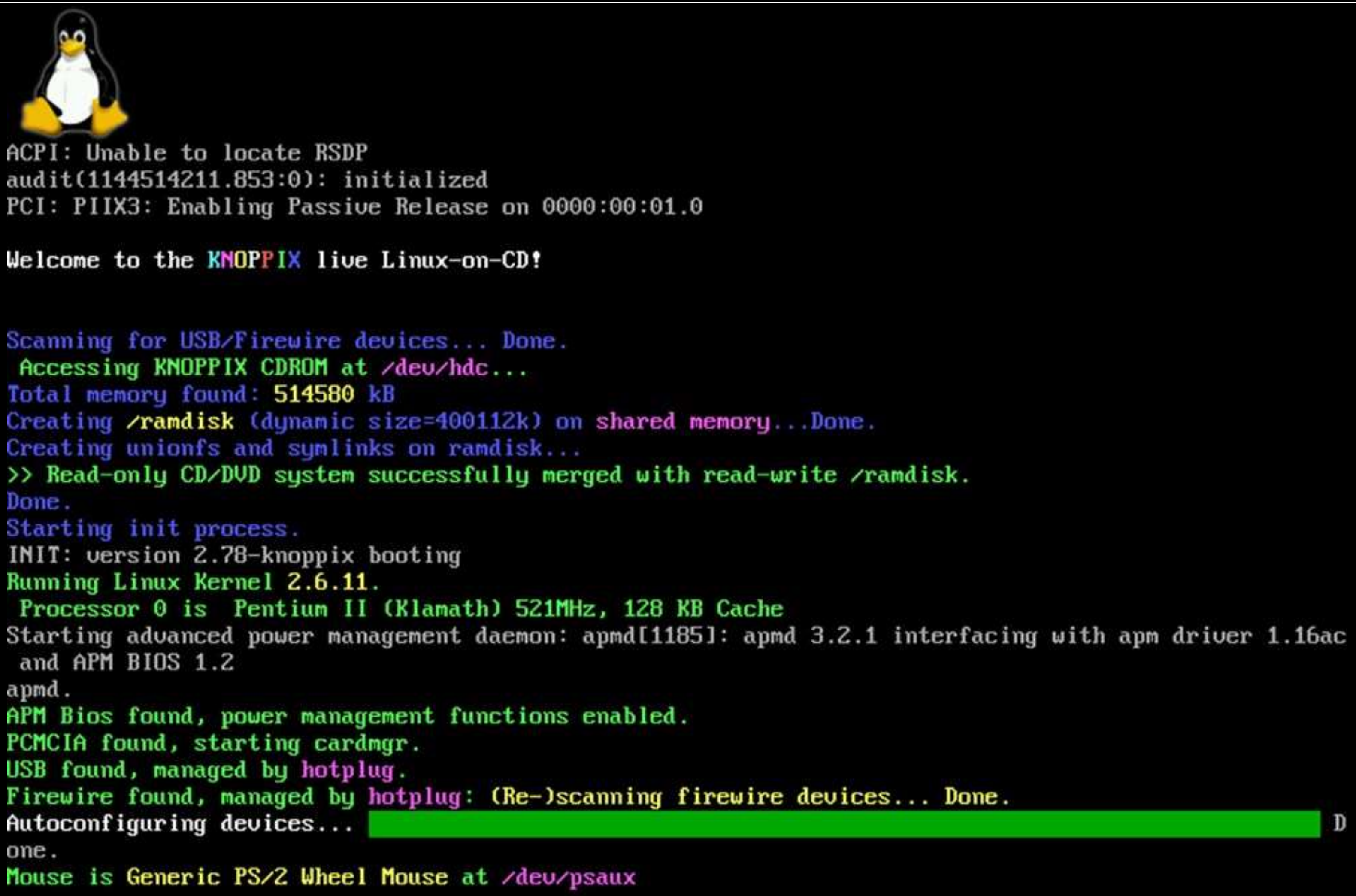
- When you logon locally, you are presented with graphical environment.
- You start at a graphical login screen. You must enter your username and password. You also the have the option to choose from a couple session types. Mainly you have the choice between Gnome and KDE.
- Once you enter in your username and password, you are then presented with a graphical environment that looks like one of the following...

UNIX interfaces

Command Line Interface

- You also have access to some UNIX servers as well.
    - You can logon from virtually any computer that has internet access whether it is Windows, Mac, or UNIX itself.
- In this case you are communicating through a local terminal to one of these remote servers.
    - All of the commands actually execute on the remote server.
    - It is also possible to open up graphical applications through this window, but that requires a good bit more setup and software.

System Boot up

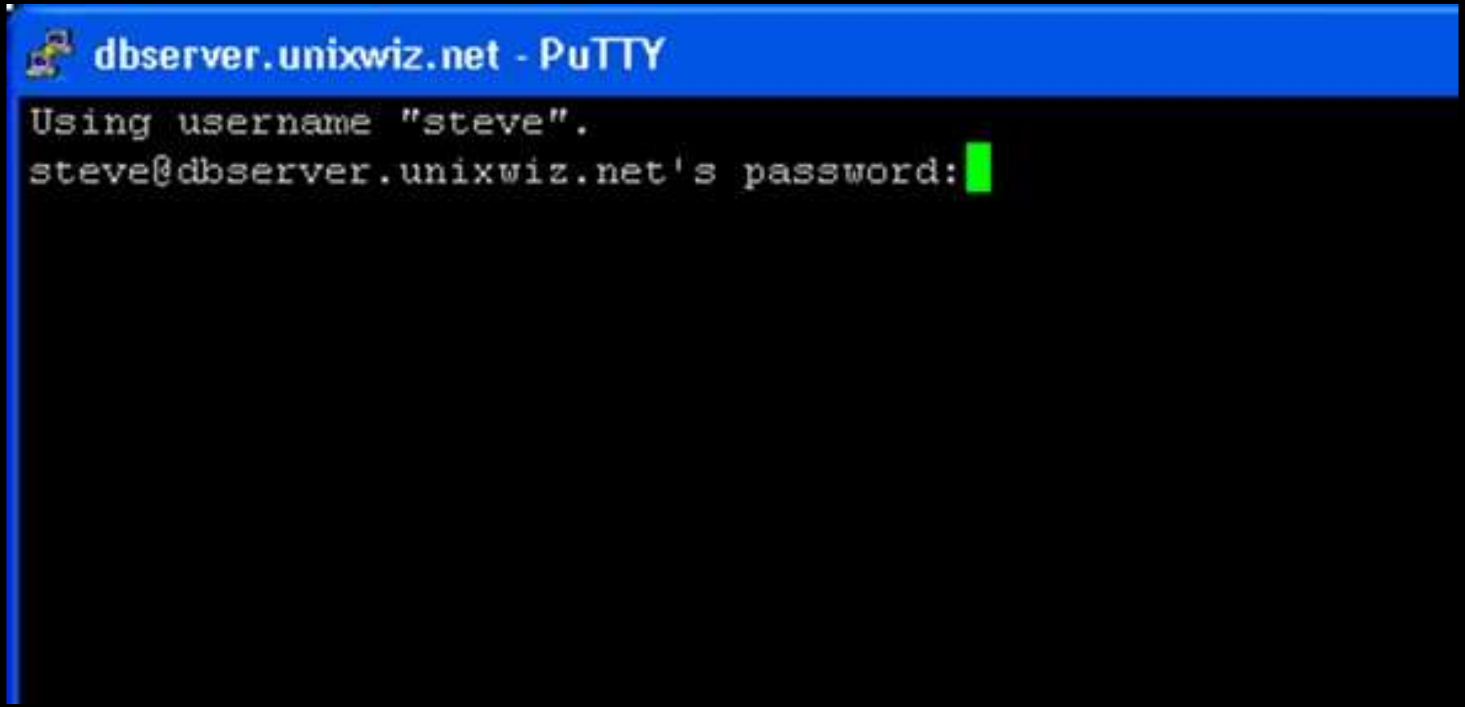


```
ACPI: Unable to locate RSDP
audit(1144514211.853:0): initialized
PCI: PIIX3: Enabling Passive Release on 0000:00:01.0

Welcome to the KNOPPIX live Linux-on-CD!


Scanning for USB/Firewire devices... Done.
 Accessing KNOPPIX CDROM at /dev/hdc...
Total memory found: 514580 kB
Creating /ramdisk (dynamic size=400112k) on shared memory...Done.
Creating unionfs and symlinks on ramdisk...
>> Read-only CD/DVD system successfully merged with read-write /ramdisk.
Done.
Starting init process.
INIT: version 2.78-knoppix booting
Running Linux Kernel 2.6.11.
 Processor 0 is  Pentium II (Klamath) 521MHz, 128 KB Cache
Starting advanced power management daemon: apmd[1185]: apmd 3.2.1 interfacing with apm driver 1.16ac
 and APM BIOS 1.2
apmd.
APM Bios found, power management functions enabled.
PCMCIA found, starting cardmgr.
USB found, managed by hotplug.
Firewire found, managed by hotplug: (Re-)scanning firewire devices... Done.
Autoconfiguring devices...                                                    D
one.
Mouse is Generic PS/2 Wheel Mouse at /dev/psaux
```

Login to UNIX



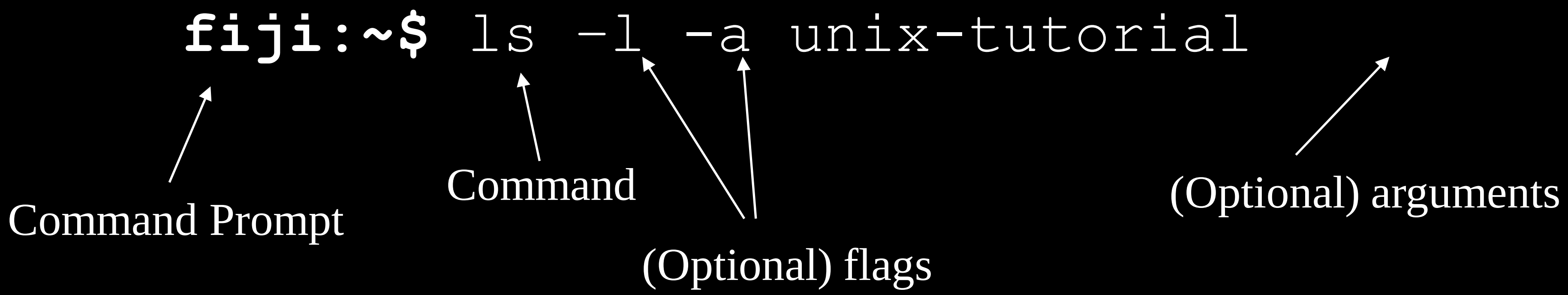
dbserver.unixwiz.net - PuTTY

```
Using username "steve".
steve@dbserver.unixwiz.net's password:
```

Command prompt

- Commands are the way to “do things” in Unix
- A command consists of a command name and options called “flags”
- Commands are typed at the *command prompt*
- In Unix, *everything* (including commands) is case-sensitive

```
[prompt]$ <command> <flags> <args>
```



```
fiji:~$ ls –l -a unix-tutorial
```

Getting started

`[prompt]$` help

To get help on any command

Getting started

Display calendar

```
$ Cal
```

Who are you?

```
$ Whoami
```

Who’s logged in

```
$ Users
$ Who
$ w
```

Logout

```
$ logout
```

```
halt
shutdown
reboot
poweroff
```

UNIX file system

- All data is organized into files
- All files are organized into directories.
- All directories are organized into a tree-like structure called the filesystem.
- **Ordinary Files** –

  Contains data, text, or program instructions.

- **Directories** –

  Directories store both special and ordinary files like folders in Windows

- **Special Files** –

  Access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters.

File management

File Access Modes -

- Read:
  - Grants the capability to read ie. view the contents of the file.
- Write:
  - Grants the capability to modify, or remove the content of the file.
- Execute:
  - User with execute permissions can run a file as a program.

File management

Directory Access Modes -

- Read:
  - Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.
- Write:
  - Access means that the user can add or delete files to the contents of the directory.
- Execute:
  - Executing a directory doesn't really make a lot of sense so think of this as a navigation permission.
  - A user must have execute access to the ***<u>bin</u>*** directory in order to execute or command.

File management

Starting a Process -

- Foreground Processes:
  - By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

- Background Processes:
  - A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.
  - The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

File management

Stopping Processes –

- Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

- If a process is running in background mode then first you would need to get its Job ID using ***ps*** command and after that you can use ***kill*** command to kill the process.

Advantages of UNIX

- Highly efficient level when it comes to the virtual memory. It simply means that you can utilize several programs while at the same time using only minimal level of physical memory.
- Variety of utilities and commands that are specially designed to perform specific tasks. With a properly stocked toolbox, UNIX can handle tasks flawlessly.
- UNIX is readily available for different machines. It is considered as a highly portable operating system since it can be used on either PC or Macintosh computers or other computing machines as well.
- As an operating system, UNIX offers the capability to run various commands and utilities in diverse configurations in order to perform complex tasks.

Disadvantages of UNIX

- Command-line based interface - makes it difficult for casual users. It is a known fact that UNIX was created to be used by programmers. There is a graphical user interface (GUI) available but the traditional UNIX interface is available as command-line only.
- Due to the productivity of the utilities of UNIX, it can be too much to handle for novice-users. UNIX is not a simple operating system, thus it is not suitable for novice users.
- The special commands needed by the command-line interface typically involve perplexing naming schemes and do not provide enough information on what the user is doing. Understandably, majority of the commands used in UNIX entail the use of special characters where minor mistakes can cause unanticipated results on the UNIX machines.

Working with files

```
[prompt]$ ls
```

To list the files and directories stored in the current directory.

Working with files

```
[prompt]$ ls -l
```

To list the files and directories with more information

- First Column: represents file type and permission given on the file. Below is the description of all type of files.
- Second Column: represents the number of memory blocks taken by the file or directory.
- Third Column: represents owner of the file. This is the Unix user who created this file.
- Fourth Column: represents group of the owner. Every Unix user would have an associated group.
- Fifth Column: represents file size in bytes.
- Sixth Column: represents date and time when this file was created or modified last time.
- Seventh Column: represents file or directory name.

Working with files

```
[prompt]$ ls re*.txt
```

Displays all the files whose name start with re and ends with .txt

```
[prompt]$ ls *.txt
```

Displays all the files whose name ends with .txt

```
[prompt]$ ls -a
```

Displays hidden files and all files in the current directorty

Single dot . – This represents current directory.

Double dot .. – This represents parent directory.

Working with files

Creating files

```
[prompt]$ vi <filename>
```

Press key esc to come out of edit mode.

Press two keys Shift + ZZ together to come out of the file completely.

Editing files

```
[prompt]$ vi <filename>
```

i key to insert or edit the file.

h key to move to the left side.

j key to move down side in the file.

l key to move to the right side.

k key to move up side in the file.

Esc to exit edit mode

Working with files

Display contents of a file

```
[prompt]$ cat <filename>
```

```
[prompt]$ cat –b <filename>
```

*You can display line numbers by using **-b** option*

```
[prompt]$ cat <filename> | more
```

You can display contents of a large file with a page break using more option

Press space key to move to next page

Press enter key to move line by line after the break

Working with files

Counting words in a file

```
[prompt]$ wc <filename>
```

- *First Column: represents total number of lines in the file.*
- *Second Column: represents total number of words in the file.*
- *Third Column: represents total number of bytes in the file. This is actual size of the file.*
- *Fourth Column: represents file name.*

- `[prompt]$ wc <filename1> <filename2>`

Working with files

Display first 10 lines of a file

```
[prompt]$ head <filename>


[prompt]$ head –n <filename>
```

Displays first n number of lines specified by n.

Display last 10 lines of a file

```
[prompt]$ tail <filename>
[prompt]$ tail –n <filename>
```

Displays last n number of lines specified by n.

Working with files

Copying files

```
[prompt]$ cp source_file destination_file
```

E.g.

```
[prompt]$ cp README.txt myfile.txt
[prompt]$ cp README.txt myfile
[prompt]$ cp myfile myfile.doc
```

Working with files

Renaming files

```
[prompt]$ mv old_filename new_filename
```

E.g.

```
[prompt]$ mv README.txt myfile.txt
[prompt]$ mv myfile.txt testfile.txt
```

Working with files

Deleting files

```
[prompt]$ rm filename
[prompt]$ rm filename1 filename2
```

Directory management

A directory is a file whose sole job is to store file names and related information. All files, whether ordinary, special, or directory, are contained in directories.

UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character (/), and all other directories are contained below it.

Directory management

Home directory

```
[prompt]$ cd ~
```

Display current directory

```
[prompt]$ pwd
```

List directories in a path

```
[prompt]$ ls /usr/local
```

Directory management

Create directory

```
[prompt]$ mkdir <dirname>
[prompt]$ mkdir mydir
```

```
[prompt]$ mkdir /tmp/test-dir
```

To create a directory in a specified path . For e.g. create test-dir in tmp directory

```
[prompt]$ mkdir <dirname1> <dirname2>
[prompt]$ mkdir docs pubs
```

To create more than one directory at one go

Directory management

Create parent directories

```
[prompt]$ mkdir /tmp/test1/mydir
```

Throws an error if the directory test1 does not exist

```
[prompt]$ mkdir –p /tmp/test1/mydir
```

This creates all the necessary directories without any problems

Directory management

Remove directories

```
[prompt]$ rmdir <dirname>

[prompt]$ rmdir <dirname1> <dirname2>
```

Directory management

Change directories

```
[prompt]$ cd <dirname>

[prompt]$ cd /usr/local
[prompt]$ cd ~
[prompt]$ cd ../
```

Directory management

Rename directories

```
[prompt]$ mv old_dir new_dir

For e.g.
[prompt]$ mv test1 test2
```

Directory management

Directories .(dot) and ..(dot dot)

The filename . (dot) represents the current working directory; and the filename .. (dot dot) represent the directory one level above the current working directory, often referred to as the parent directory.

```
[prompt]$ ls -la
```

Directory management

Copy files between directories

```
[prompt]$ cp README.txt /tmp/test1/myfile.txt

[prompt]$ cd /tmp/test1
[prompt]$ ls -l
```

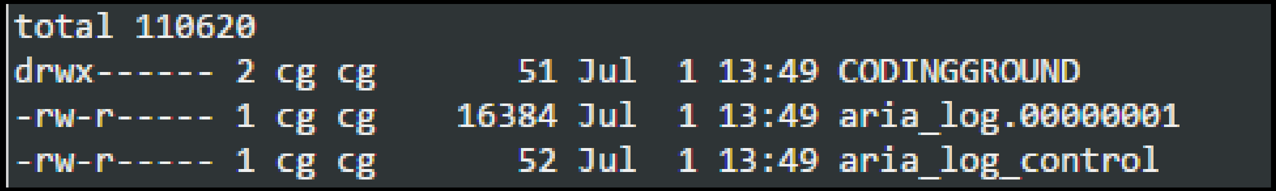
File permission / access modes

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes −

- Owner permissions − The owner's permissions determine what actions the owner of the file can perform on the file.
- Group permissions − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- Other (world) permissions − The permissions for others indicate what action all other users can perform on the file.

File permission / access modes

```
[prompt]$ ls –l /home/cg/mysql
```



```
total 110620
drwx------ 2 cg cg       51 Jul  1 13:49 CODINGGROUND
-rw-r----- 1 cg cg    16384 Jul  1 13:49 aria_log.00000001
-rw-r----- 1 cg cg       52 Jul  1 13:49 aria_log_control
```

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that onwer has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--** represents that group has read (r) and execute (x) permission but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

File permission / access modes

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the read, write, and execute permissions, which are described below −

1. Read

Grants the capability to read ie. view the contents of the file.

2. Write

Grants the capability to modify, or remove the content of the file.

3. Execute

User with execute permissions can run a file as a program.

File permission / access modes

Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

1. Read

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

2. Write

Access means that the user can add or delete files to the contents of the directory.

3. Execute

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.

A user must have execute access to the bin directory in order to execute ls or cd command.

File permission / access modes

Changing permissions

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

```
[prompt]$ ls -l testfile
-rwxrwxr-- 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

```
[prompt]$ ls –l testfile
-rwxrwxr–- 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod o+wx testfile
[prompt]$ ls -l testfile
-rwxrwxrwx 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

```
[prompt]$ chmod u-x testfile
[prompt]$ ls -l testfile
-rw-rwxrwx 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod g=rx testfile
[prompt]$ ls -l testfile
-rw-r-xrwx 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

Adding permissions together in one command

```
[prompt]$ ls -l testfile
-rw-r-xrwx 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod o-w, g=rwx, u+x testfile
[prompt]$ ls -l testfile
-rwxrwxr-x 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

File permission / access modes

Changing permissions

```
[prompt]$ ls -l testfile
-rwxrwxr-- 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod 755 testfile
[prompt]$ ls -l testfile
-rwxr-xr-x 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

```
[prompt]$ ls -l testfile
-rwxr-xr-x 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod 743 testfile
[prompt]$ ls -l testfile
-rwxr---wx 1 cgusr users 1024 Sep 3 00:00 testfile
```

File permission / access modes

Changing permissions

```
[prompt]$ ls -l testfile
-rwxr---wx 1 cgusr users 1024 Sep 3 00:00 testfile

[prompt]$ chmod 043 testfile
[prompt]$ ls -l testfile
---r---wx 1 cgusr users 1024 Sep 3 00:00 testfile
```

UNIX Environment

PS1 and PS2 Variables

```
[prompt]$ PS1='=>'
=>ls -l filename
=>

=>PS1="[\u@\h \w]$"
```

UNIX Environment

Escape Sequence	Description
\t	Current time, expressed as HH:MM:SS.
\d	Current date, expressed as Weekday Month Date
\n	Newline.
\s	Current shell environment.
\W	Working directory.
\w	Full path of the working directory.
\u	Current user.s username.
\h	Hostname of the current machine.
\#	Command number of the current command. Increases with each new command entered.
\$	If the effective UID is 0 (that is, if you are logged in as root), end the prompt with the # character; otherwise, use the $.

UNIX Environment

PS2 Variable

```
=> echo "this is a
> test"
this is a
test
=>
=> PS2="secondary prompt->"
=> echo "this is a
secondary prompt->test"
```

UNIX Environment

Some useful environment variables

```
[prompt]$ echo $HOME
[prompt]$ echo $DISPLAY
[prompt]$ echo $PATH
[prompt]$ echo $RANDOM
[prompt]$ df
[prompt]$ du /home
```

Pipes and Filters

You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.

To make a pipe, put a vertical bar (|) on the command line between two commands.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a *filter*.

Pipes and Filters

grep command

```
[prompt]$ ls –l | grep "Sep"
-rw-rw-rw-   1 john  doc     11008 Sep 6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Sep 6 15:30 ch07
```

Pipes and Filters

grep command

Option	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

```
[prompt]$ ls –l | grep –i “read*.*” | grep –i “sep”
-rw-rw-r--   1 README txt     1605 Sep 2 07:35 macros
```

Pipes and Filters

grep command

Option	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

```
[prompt]$ ls –l | grep –i “read*.*” | grep –i “sep”
-rw-rw-r--   1 README txt     1605 Sep 2 07:35 macros
```

Pipes and Filters

Sort command

```
Create a file called “food”
[prompt]$ vi food
Idli
Vada
Dosa
Samosa
Kachori
Roti
Batura
```

Pipes and Filters

Sort command

```
[prompt]$ cat food
Idli
Vada
Dosa
Samosa
Kachori
Roti
Batura
```

Pipes and Filters

Sort command

```
[prompt]$ sort food
Batura
Dosa
Idli
Kachori
Roti
Samosa
Vada
```

Pipes and Filters

Sort command

The sort command arranges lines of text alphabetically by default. There are many options that control the sorting –

Option	Description
-n	Sort numerically (example: 10 will sort after 2), ignore blanks and tabs.
-r	Reverse the order of sort.
-f	Sort upper- and lowercase together.
+x	Ignore first x fields when sorting.

The vi editor

There are many ways to edit files in Unix and for me one of the best ways is using screen-oriented text editor vi. This editor enable you to edit lines in context with other lines in the file.

Now a days you would find an improved version of vi editor which is called VIM. Here VIM stands for Vi IMproved.

The vi editor

The vi is generally considered the de facto standard in Unix editors because −

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user friendly than any other editors like ed or ex.

You can use vi editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

The vi editor

There are following way you can start using vi editor –

Command	Description
vi filename	Creates a new file if it already does not exist, otherwise opens existing file.
vi -R filename	Opens an existing file in read only mode.
view filename	Opens an existing file in read only mode.

```
[prompt]$ vi testfile
```

The vi editor

There are following way you can start using vi editor –

```
|
~
~
.
.
.
.
"testfile" [New File]
```

You will notice a tilde (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other nonviewable character present.

The vi editor

While working with vi editor you would come across following two modes

Command mode − This mode enables you to perform administrative tasks such as saving files, executing commands, moving the cursor, cutting (yanking) and pasting lines or words, and finding and replacing. In this mode, whatever you type is interpreted as a command.

Insert mode − This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and finally it is put in the file .

The vi always starts in command mode. To enter text, you must be in insert mode. To come in insert mode you simply type i. To get out of insert mode, press the Esc key, which will put you back into command mode.

Hint − If you are not sure which mode you are in, press the Esc key twice, and then you'll be in command mode. You open a file using vi editor and start type some characters and then come in command mode to understand the difference.

The vi editor

Exiting vi

:q – *quit with prompt to save file*

:q! – *quit without prompt to save file*

:w – *save file*

Shift+ZZ – *equal to :wq where the file is saved and quit from vi*

:w *<filename>* - *saves to another filename*

The vi editor

Moving within the file

Command	Description
k	Moves the cursor up one line.
j	Moves the cursor down one line.
h	Moves the cursor to the left one character position.
l	Moves the cursor to the right one character position.

There are following two important points to be noted −

The vi is case-sensitive, so you need to pay special attention to capitalization when using commands.

Most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2j moves cursor two lines down the cursor location.

The vi editor

Command	Description
0 or |	Positions cursor at beginning of line.
$	Positions cursor at end of line.
w	Positions cursor to the next word.
b	Positions cursor to previous word.
(	Positions cursor to beginning of current sentence.
)	Positions cursor to beginning of next sentence.
E	Move to the end of Blank delimited word
{	Move a paragraph back
}	Move a paragraph forward
[[	Move a section back
]]	Move a section forward
n|	Moves to the column n in the current line

Command	Description
1G	Move to the first line of the file
G	Move to the last line of the file
nG	Move to nth line of the file
:n	Move to nth line of the file
fc	Move forward to c
Fc	Move back to c
H	Move to top of screen
nH	Moves to nth line from the top of the screen
M	Move to middle of screen
L	Move to botton of screen
nL	Moves to nth line from the bottom of the screen
:x	Colon followed by a number would position the cursor on line number represented by **x**

The vi editor

Control commands

Command	Description
CTRL+d	Move forward 1/2 screen
CTRL+d	Move forward 1/2 screen
CTRL+f	Move forward one full screen
CTRL+u	Move backward 1/2 screen
CTRL+b	Move backward one full screen
CTRL+e	Moves screen up one line
CTRL+y	Moves screen down one line
CTRL+u	Moves screen up 1/2 page
CTRL+d	Moves screen down 1/2 page
CTRL+b	Moves screen up one page
CTRL+f	Moves screen down one page
CTRL+l	Redraws screen

The vi editor

Editing files

Command	Description
i	Inserts text before current cursor location.
I	Inserts text at beginning of current line.
a	Inserts text after current cursor location.
A	Inserts text at end of current line.
o	Creates a new line for text entry below cursor location.
O	Creates a new line for text entry above cursor location.

The vi editor

Deleting characters

Command	Description
x	Deletes the character under the cursor location.
X	Deletes the character before the cursor location.
dw	Deletes from the current cursor location to the next word.
d^	Deletes from current cursor position to the beginning of the line.
d$	Deletes from current cursor position to the end of the line.
D	Deletes from the cursor position to the end of the current line.
dd	Deletes the line the cursor is on.

As mentioned above, most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2x deletes two character under the cursor location and 2dd deletes two lines the cursor is on.

The vi editor

Copy and Paste commands

Command	Description
yy	Copies the current line.
yw	Copies the current word from the character the lowercase w cursor is on until the end of the word.
p	Puts the copied text after the cursor.
P	Puts the yanked text before the cursor.

As mentioned above, most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2x deletes two character under the cursor location and 2dd deletes two lines the cursor is on.

The vi editor

Word searching -

The / command searches forwards (downwards) in the file.

The ? command searches backwards (upwards) in the file.

Character searching -

Character	Description
^	Search at the beginning of the line. (Use at the beginning of a search expression.)
.	Matches a single character.
*****	Matches zero or more of the previous character.
$	End of the line (Use at the end of the search expression.)
[	Starts a set of matching, or non-matching expressions.
<	Put in an expression escaped with the backslash to find the ending or beginning of a word.
>	See the '<' character description above.

The vi editor

IMPORTANT

Here are the key points to your success with vi –

- You must be in command mode to use commands. (Press Esc twice at any time to ensure that you are in command mode.)
- You must be careful to use the proper case (capitalization) for all commands.
- You must be in insert mode to enter text.

HAPPY vi EDITING!!! ☺

UNIX Shell Scripting

The shell provides you with an interface to the UNIX system.

It gathers input from you and executes programs based on that input.
When a program finishes executing, it displays that program's output.

The prompt, $, which is called command prompt, is issued by the shell.

While the prompt is displayed, you can type a command.

```
[prompt]$  date
Fri Sep 3 12:30:19 MST 2015
```

UNIX Shell Scripting

In UNIX there are two major types of shells:

The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.

The C shell. If you are using a C-type shell, the default prompt is the % character.

UNIX Shell Scripting

There are again various subcategories for Bourne Shell which are listed as follows −

Bourne shell (sh)

Korn shell (ksh)

Bourne Again shell (bash)

POSIX shell (sh)

The different C-type shells follow −

C shell (csh)

TENEX/TOPS C shell (tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

UNIX Shell Scripting

There are again various subcategories for Bourne Shell which are listed as follows −

Bourne shell (sh)

Korn shell (ksh)

Bourne Again shell (bash)

POSIX shell (sh)

The different C-type shells follow −

C shell (csh)

TENEX/TOPS C shell (tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

UNIX Shell Scripting

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign, #, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

UNIX Shell Scripting

Example Script

Assume we create a ***test.sh*** script. Note all the scripts would have .sh extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the Bourne shell.
It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

UNIX Shell Scripting

Example Script

To create a script containing these commands, you put the shebang line first and then add the commands to the test.sh file

```
#!/bin/bash

pwd
ls
```

Now you save the above content and make this script executable as follows −

```
[prompt]$  chmod +x test.sh
```

Now you have your shell script ready to be executed as follows −

[prompt]$ ./test.sh *(dot forward-slash test.sh)*

UNIX Shell Scripting

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use the read command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on the screen.

```
#!/bin/sh

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"
```

Here is sample run of the script −

```
[prompt]$  ./test.sh

What is your name?

Dennis Ritchie

Hello, Dennis Ritchie
```

UNIX Shell Scripting

Shell variables

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

UNIX Shell Scripting

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix Shell variables would have their names in UPPERCASE.

Following examples are valid variable names –	Following are the examples of invalid variable names –
_ALI TOKEN_A VAR_1 VAR_2	2_VAR -VARIABLE VAR1-VAR2 VAR_A!

The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.

UNIX Shell Scripting

Defining Variables

Variables are defined as follows −

```
variable_name=variable_value
```

For example:

```
NAME=“Dennis Ritchie"
```

Above example defines the variable NAME and assigns it the value " Dennis Ritchie ".
Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The shell enables you to store any value you want in a variable. For example −

```
VAR1=“Dennis Ritchie"
VAR2=200
```

UNIX Shell Scripting

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign ($) −

For example, following script would access the value of defined variable NAME and would print it on STDOUT −

```
#!/bin/sh

NAME=“Dennis Ritchie"
echo $NAME
```

This would produce following value −

```
 Dennis Ritchie
```

UNIX Shell Scripting

Read-only Variables

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME −

```
#!/bin/sh
NAME=“Dennis Ritchie"
readonly NAME
NAME=“Thomas Edison"
```

This would produce following result −

```
/bin/sh: NAME: This variable is read only.
```

UNIX Shell Scripting

Unsetting Variables

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the unset command −

```
unset variable_name
```

Above command would unset the value of a defined variable. Here is a simple example −

```
#!/bin/sh
NAME=“Dennis Ritchie"
unset NAME
echo $NAME
```

Above example would not print anything. You cannot use the unset command to unset variables that are marked readonly.

UNIX Shell Scripting

Command-Line Arguments

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line −

```
#!/bin/sh
echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

UNIX Shell Scripting

Command-Line Arguments

Here is a sample run for the above script −

```
[prompt]$ ./test.sh Dennis Ritchie
File Name : ./test.sh
First Parameter : Dennis
Second Parameter : Ritchie
Quoted Values: Dennis Ritchie
Quoted Values: Dennis Ritchie
Total Number of Parameters : 2
```

UNIX Shell Scripting

Command-Line Arguments

```
#!/bin/sh
for TOKEN in $*
do
   echo $TOKEN
done

$./test.sh UNIX is 40 years old
UNIX
is
y0
years
old
```

UNIX Shell Scripting

Defining Arrays

```
NAME01=“Rohan"
NAME02=“Ramya"
NAME03=“Mumtaz"
NAME04=“Aryan"
NAME05=“Deepthi“

NAME[0]=“Rohan"
NAME[1]=“Ramya"
NAME[2]=“Mumtaz"
NAME[3]=“Aryan"
NAME[4]=“Deepthi“
```

UNIX Shell Scripting

Defining Arrays

```
#!/bin/sh
NAME[0]=“Rohan"
NAME[1]=“Ramya"
NAME[2]=“Mumtaz"
NAME[3]=“Aryan"
NAME[4]=“Deepthi“
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[2]}“

echo “First Method: ${[NAME[*]}”
echo “Second Method: ${[NAME[@]}”
```

```
[prompt] $./test.sh
First Index: Rohan
Second Index: Mumtaz
First Method: Rohan Ramya Mumtaz Aryan Deepthi
Second Method: Rohan Ramya Mumtaz Aryan Deepthi
```

UNIX Shell Scripting

There are various operators supported by each shell.

There are following operators which we are going to discuss −

- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

UNIX Shell Scripting

```
#!/bin/sh
val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result −

```
Total value : 4
```

UNIX Shell Scripting

Operator	Description	Example
+	Addition - Adds values on either side of the operator	\`expr $a + $b\` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	\`expr $a - $b\` will give -10
*	Multiplication - Multiplies values on either side of the operator	\`expr $a \* $b\` will give 200
/	Division - Divides left hand operand by right hand operand	\`expr $b / $a\` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	\`expr $b % $a\` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

UNIX Shell Scripting

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

This would produce following result −

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

UNIX Shell Scripting

Relational Operators

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

UNIX Shell Scripting

```
#!/bin/sh


a=10
b=20


if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi
```

```
if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi


if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi
```

UNIX Shell Scripting

```
if [ $a -lt $b ]
then
  echo "$a -lt $b: a is less than b"
else
  echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
  echo "$a -ge $b: a is greater or  equal to b"
else
  echo "$a -ge $b: a is not greater or equal to b"
fi
```

```
if [ $a -le $b ]
then
  echo "$a -le $b: a is less or  equal to b"
else
  echo "$a -le $b: a is not less or equal to b"
fi
```

UNIX Shell Scripting

The result would be:-

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

UNIX Shell Scripting

Boolean Operators

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

UNIX Shell Scripting

```
#!/bin/sh
a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi
```

```
if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

UNIX Shell Scripting

```
if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

This would produce following results −

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

UNIX Shell Scripting

String Operators

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

UNIX Shell Scripting

```
#!/bin/sh
a="abc"
b="efg"

if [ $a = $b ]
then
  echo "$a = $b : a is equal to b"
else
  echo "$a = $b: a is not equal to b"
fi
```

```
if [ $a != $b ]
then
  echo "$a != $b : a is not equal to b"
else
  echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
  echo "-z $a : string length is zero"
else
  echo "-z $a : string length is not zero"
fi
```

UNIX Shell Scripting

```
if [ -n $a ]
then
  echo "-n $a : string length is not zero"
else
  echo "-n $a : string length is zero"
fi

if [ $a ]
then
  echo "$a : string is not empty"
else
  echo "$a : string is empty"
fi
```

This would produce following result −

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

UNIX Shell Scripting

Decision Making

- The if...else statements
- The case...esac statement

UNIX Shell Scripting

The if...else statements

Unix Shell supports following forms of if..else statement –

if...fi statement

if...else...fi statement

if...elif...else...fi statement

UNIX Shell Scripting

if...fi statement

Syntax -

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

UNIX Shell Scripting

if...fi statement

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

This will produce following result −

```
a is not equal to b
```

UNIX Shell Scripting

if...fi statement

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
  echo "a is equal to b"
fi

if [ $a != $b ]
then
  echo "a is not equal to b"
fi
```

This will produce following result −

```
a is not equal to b
```

UNIX Shell Scripting

if...else...fi statement

Syntax -

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

UNIX Shell Scripting

if...else...fi statement

```
#!/bin/sh

a=10
b=20

if [ $a > $b ]
then
   echo "a is greater than b"
else
   echo "a is lesser than b"
fi
```

This will produce following result −

```
a is lesser than b
```

UNIX Shell Scripting

if...elif...else...fi statement

Syntax -

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]…
else
   Statement(s) to be executed if no expression is true
fi
```

UNIX Shell Scripting

if...elif...else...fi statement

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
  echo "a is equal to b"
elif [ $a -gt $b ]
then
  echo "a is greater than b"
elif [ $a -lt $b ]
then
  echo "a is less than b"
else
  echo "None of the condition met"
fi
```

This will produce following result −

```
a is lesser than b
```

UNIX Shell Scripting

case...esac statement

Syntax –

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

UNIX Shell Scripting

case...esac statement

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

This will produce following result −

```
New Zealand is famous for kiwi.
```

UNIX Shell Scripting

case...esac statement

```
#!/bin/sh
option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Here is a sample run of this program −

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

UNIX Shell Scripting

Loop types

- The while loop
- The for loop
- The until loop
- The select loop

UNIX Shell Scripting

The while loop

Syntax –

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

UNIX Shell Scripting

The while loop

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

UNIX Shell Scripting

The for loop

Syntax –

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

UNIX Shell Scripting

The for loop

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
do
  echo $var
done
```

```
#!/bin/sh

for FILE in $HOME/.bash*
do
  echo $FILE
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

UNIX Shell Scripting

The until loop

Syntax –

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

UNIX Shell Scripting

The until loop

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result −

```
0
1
2
3
4
5
6
7
8
9
```

UNIX Shell Scripting

The select loop

Syntax –

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

UNIX Shell Scripting

The select loop

```
#!/bin/ksh
select DRINK in tea cofee water juice appe all none
do
  case $DRINK in
    tea|cofee|water|all)
      echo "Go to canteen"
      ;;
    juice|appe)
      echo "Available at home"
    ;;
    none)
      break
    ;;
    *) echo "ERROR: Invalid selection"
    ;;
  esac
done
```

The menu presented by the select loop looks like the following –

```
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
#? juice
Available at home
#? none
$
```

UNIX Shell Scripting

Nested loops

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

The menu presented by the select loop looks like the following −

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

UNIX Shell Scripting

Nested loops

Syntax –

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done
   Statement(s) to be executed if command1 is true
done
```

UNIX Shell Scripting

Quoting mechanisms

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example ? matches with a single charater while listing files in a directory and an * would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters) −

* ? [ ] ' " \ $; & ( ) | ^ < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

UNIX Shell Scripting

Example

Following is the example which show how to print a * or a ? −

```
#!/bin/sh

echo Hello; Word
```

This would produce following result −

```
Hello
./test.sh: line 2: Word: command not found

shell returned 127
```

UNIX Shell Scripting

Example

Now let us try using a quoted character −

```
#!/bin/sh
echo Hello\; Word
```

This would produce following result −

```
Hello; Word
```

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell −

```
#!/bin/sh
echo "I have \$1200"
```

This would produce following result −

```
I have $1200
```

UNIX Shell Scripting

There are following four forms of quotings −

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	•Most special characters between these quotes lose their special meaning with these exceptions: $ •` •\$ •\' •\" •\\
Backslash	Any character immediately following the backslash loses its special meaning.
Back Quote	Anything in between back quotes would be treated as a command and would be executed.

UNIX Shell Scripting

The Single Quotes

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read −

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string −

```
echo '<-$1500.**>; (update?) [y|n]'
```

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

```
echo 'It\'s Shell Programming'
```

UNIX Shell Scripting

The Double Quotes

```
VAR=DENNIS
echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

This would produce following result −

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows −

```
VAR=DENNIS
echo "$VAR owes <-\$1500.**>; [ as of (`date +%m/%d`) ]"
```

Now this would produce following result −

```
DENNIS owes <-$1500.**>; [ as of (07/02) ]
```

UNIX Shell Scripting

The Back Quotes

Putting any Shell command in between back quotes would execute the command

```
DATE=`date`
```

```
echo "Current Date: $DATE"
```

This would produce following result −

```
Current Date: Thu Jul  2 05:28:45 MST 2009
```

UNIX Shell Scripting

Output redirection

```
$ who > users

$ cat users
oko        tty01   Sep 12 07:30
ai         tty15   Sep 12 13:32
ruth       tty21   Sep 12 10:10
pat        tty24   Sep 12 13:07
steve      tty25   Sep 12 13:03
$
```

UNIX Shell Scripting

Output redirection

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

UNIX Shell Scripting

Here command

Syntax –

```
command << delimiter
document
delimiter
```

```
#!/bin/sh
cat  << EOF
This is a simple lookup program
for a code that is input
automatically from the code.
EOF
```

This would produce following result −

```
This is a simple lookup program
for a code that is input
automatically from the code.
```

UNIX Shell Scripting

The following script runs a session with the vi text editor and save the input in the file test.txt.

```
#!/bin/sh
filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

This would produce following result −

If you run this script with vim acting as vi, then you will likely see output like the following −

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt −

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

UNIX Shell Scripting

Loop control with “break” and “continue”

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
  echo $a
  if [ $a -eq 5 ]
  then
    break
  fi
  a=`expr $a + 1`
done
```

UNIX Shell Scripting

Loop control with “break” and “continue”

```
#!/bin/sh
for var1 in 1 2 3
do
  for var2 in 0 5
  do
    if [ $var1 -eq 2 -a $var2 -eq 0 ]
    then
      break 2
    else
      echo "$var1 $var2"
    fi
  done
done
```

UNIX Shell Scripting

Loop control with “break” and “continue”

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7"
for NUM in $NUMS
do
  Q=`expr $NUM % 2`
  if [ $Q -eq 0 ]
  then
    echo "Number is an even number!!"
    continue
  fi
  echo "Found odd number"
done
```