Introduction to UNIX

What is your skill level?

- How do you display the first 3 lines of logfile?
- How do you remove the 7th line from a file?
- What is Unix pipe?
- What does 2>&1   do?
- What is the output of "`grep root /etc/password | grep -v "No such file or directory`"
- How to find process which is taking maximum memory in server?
- What does "`ls * ; echo *`" so ?
- What is relative path and absolute path
- What is the difference between "echo $PATH" and echo "${PATH}"
- What is wild-card interpretation?
- What does the below command do?
- What is the output : `$who | sort –logfile > newfile`
- How will you count number of occurrence of a word in a file?
- How will you replace a word (e.g "bad" with "good") in a file? What if the file is 1GB with 1Million lines?

About This Course

Course Goal

- Hands on with Unix command
- Develop shell scripts

Linux Distributions

debian

ubuntu

Live GNU/Linux System

KNOPPIX

DSL

50MB Live BizCard CD

redhat.

fedora

CentOS

Scientific Linux

slackware

linux

gentoo linux

slax

openSUSE

Unix Principles

- ✓ Everything is a file. Including hardware!!
- ✓ Configuration data stored in text (Store data and configuration info in flat ASCII files)
- ✓ Small, single-purpose programs
- ✓ Avoid Captive User Interfaces
- ✓ Ability to chain programs together to perform complex tasks.
- ✓ Make each program do one thing well
- ✓ Small is beautiful
- ✓ Choose portability over efficiency
- ✓ Use shell scripts to increase leverage and portability

http://en.wikipedia.org/wiki/Unix_philosophy

Unix Quotes..

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity. -- Dennis Ritchie

"The number of UNIX installations has grown to 10, with more expected." Unix Programmer's Manual, 2nd Ed., June 12, 1972

UNIX was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things. -- unknown

"Be afraid, be very afraid for you have root access" – unknown

Contrary to popular belief, Unix is user friendly. It just happens to be very selective about who it decides to make friends with. -- unknown

The UNIX legacy is a set of simple and timeless tools that can take years to master but which can perform seeming miracles in seconds in the hands of experienced users. -- Linux Users Group member

Unix Quotes..

I think the major good idea in Unix was its clean and simple interface: open, close, read, and write. - Ken Thompson

Unix gives you just enough rope to hang yourself -- and then a couple of more feet, just to be sure. (Quote by - Eric Allman)

"To err is human... to really foul up requires the root password.“

"If brute force doesn't solve your problems, then you aren't using enough.“

I got tired of people complaining that it was too hard to use UNIX because the editor was too complicated. (Quote by - Bill Joy)

"It is not UNIX's job to stop you from shooting your foot. If you so choose to do so, then it is UNIX's job to deliver Mr. Bullet to Mr Foot in the most efficient way it knows." -- Terry Lambert

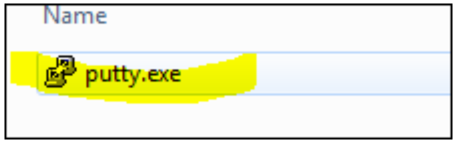
The root user

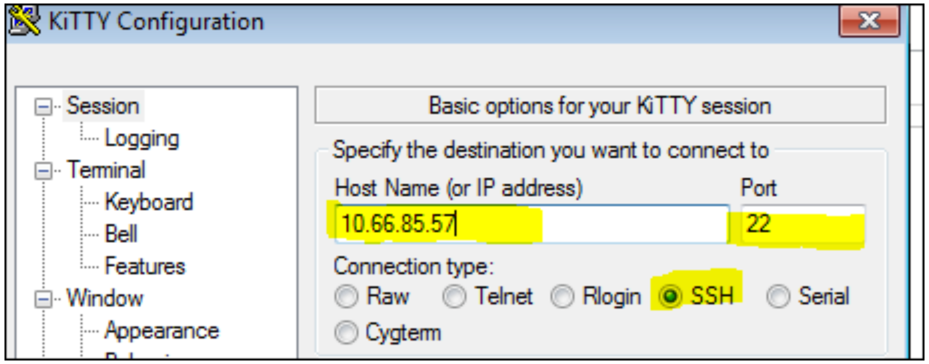
- ✓ The root user: a special administrative account
- ✓ Also called the superuser

- ✓ root has near complete control over the system and a nearly unlimited capacity to damage it....!
- ✓ Do not login as root unless necessary
- ✓ Normal users potential to do damage is more limited

- ✓ su - creates new shell as root
- ✓ sudo command runs command as root
- ✓ Requires prior configuration by a system administrator
- ✓ id shows information on the current user

Accessing Infy Lab


Name
putty.exe


KiTTY Configuration

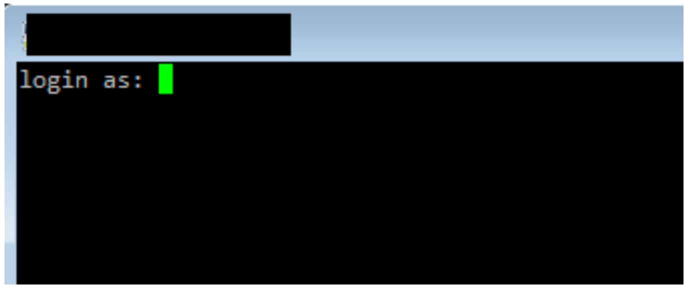
Basic options for your KiTTY session

Specify the destination you want to connect to

Host Name (or IP address): 10.66.85.57

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial Cygterm


[redacted]

```
login as: 
```

Check Connectivity…

```
UNIX User ID detail:

Username:             user10 ... user40
Password:               #infy123
IpAddress:              10.66.85.57
```

Shell…

Shell is a Unix term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters on operating system.

Another explanation:

A Unix shell, also called "the command line", provides the traditional user interface for the Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering command input as text for a shell to execute

Source : Wikipedia

Shell types…

- ✓ Bourne shell (sh)
- ✓ C shell (csh)
- ✓ TC shell (tcsh)
- ✓ Korn shell (ksh)
- ✓ Bourne Again SHell (bash)

Shell commands…

They are part of a shell, each shell (C Shell, Bourne Shell and Korn Shell) has a set of commands built into its program, the commands within each shell stay the same across Linux / Unix distributions

```
$ type cd

To see the usage:
$ help cd
```

Everyday Built-in shell commands….

```
help
cd
pwd
alias/unalias
break
for
echo
export
fg
history
kill
read
printf
test
type
wait
```

Lot more…. Check the man pages

How the shell works… what happens when you run a command

The command (e.g 'ls'), doesn't fork.

The shell forks (e.g. 'ls') and execs in order to run any command that isn't built in.

The shell is forked even for scripts

Try this (mycd.sh) and see if t works:

```
#!/bin/bash
cd /tmp
```

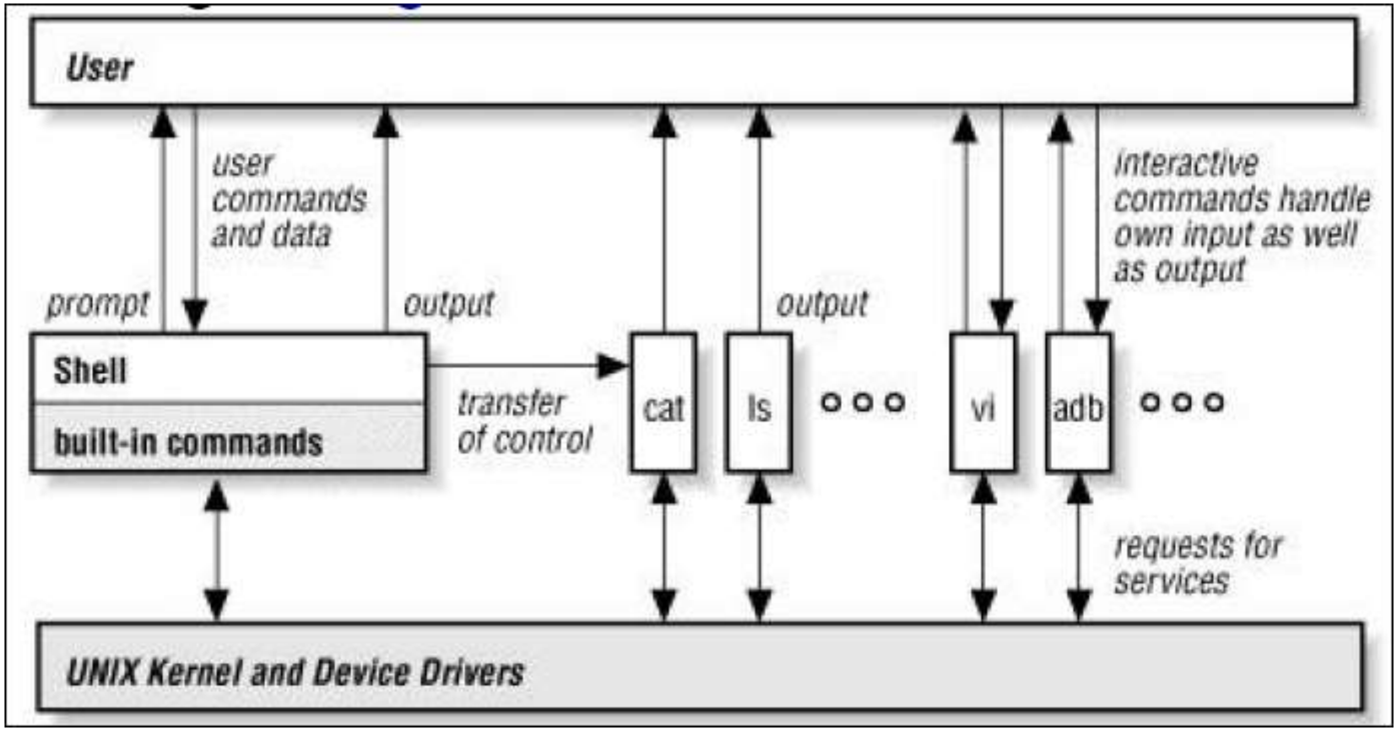
Tip: put "pwd" to see if "cd" really works in the script or not!!

Note: We will see later how to make this work ☺

How the shell works… what happens when you run a command

Shell scripts are also run inside a subshell, and each subshell has its own environment of what the current directory is. The cd succeeds, but as soon as the subshell exits, you're back in the interactive shell and nothing ever changed there

Relationship of kernel, shell, utilities…



Source : Solaris Admin Guide

How the shell works… make sure the environment is passed…

For all scripts and CRONTAB jobs, its very important that we make sure that the complete environment is passed on to make sure that it works properly.

Do not assume anything

Command line

All command line programs have 3 main components:

```
Command line arguments
Standard Input (stdin)
Standard Output (stdout)
```

By default, stdin is typed from the terminal and stdout is printed to the terminal

```
$ sed 's/simple/awesome/g'
This example is simple
This example is awesome
^D
```

Command line

```
$ cat > file.txt
Some random stuff...
^D

$ cat < file.txt
Some random stuff...


$ cat *.txt
Some random stuff...


$ cat *.txt | sort | uniq > out.txt
```

Command line

```
$ cat > file.txt
Some random stuff...
^D

$ cat < file.txt
Some random stuff...


$ cat *.txt
Some random stuff...


$ cat *.txt | sort | uniq > out.txt
```

Command line

```
$ cat file.txt| tee output.txt
Some random stuff...

$ cat file.txt| less
Some random stuff...



$ ls -l
-rw-r--r-- 1 binny staff 21 Feb  3 14:17 file.txt
-rw-r--r-- 1 binny staff 21 Feb  3 14:17 out.txt
-rw-r--r-- 1 binny staff 21 Feb  3 14:18 output.txt


$ ls -l | awk '{s+=$5} END{print s}`
63
```

Command line

```
$ cat file.txt| tee output.txt
Some random stuff...

$ cat file.txt| less
Some random stuff...


$ ls -l
-rw-r--r-- 1 binny staff 21 Feb  3 14:17 file.txt
-rw-r--r-- 1 binny staff 21 Feb  3 14:17 out.txt
-rw-r--r-- 1 binny staff 21 Feb  3 14:18 output.txt


$ ls -l | awk '{s+=$5} END{print s}'
63
```

Standard I/O and Pipes

- Standard Input and Output
- Linux provides three I/O channels to Programs

- Standard input (STDIN) - keyboard by default
- Standard output (STDOUT) - terminal window by default
- Standard error (STDERR) - terminal window by default

Standard I/O and Pipes

- Redirecting Output....
- STDOUT and STDERR can be redirected
- > Redirect STDOUT to file
- 2> Redirect STDERR to file
- &> Redirect all output to file
- 2>&1 Redirects STDERR to STDOUT

```
$ find /etc -name passwd > find.out
$ find /etc -name passwd 2> /dev/null
$ find /etc -name passwd > find.out 2> find.err
```

- File contents are overwritten by default. >> appends

```
$ find /home >> find.out
```

Standard I/O and Pipes

- Combining Output and Errors
- &>: Redirects all output:

```
$ find /etc -name passwd &> find.all
```

- 2>&1: Redirects STDERR to STDOUT
  (Useful for sending all output through a pipe)

```
$ find  /abc /etc 2>&1  | grep passwd
$ find  /abc /etc 2>&1  | grep file
```

- Combines STDOUTs of multiple programs using ‘(‘ and ‘)’

```
$ ( cal 2015; cal 2016 ) | less
```

Standard I/O and Pipes

- Redirecting to Multiple Targets (tee)

```
$ ls -l | tee listing | wc -l
```

- Stores STDOUT of “ls” in “listing” , then pipes to “wc”
- Typically used for Troubleshooting complex pipelines and Simultaneous viewing and logging of output

- Redirecting STDIN from a File (Redirect standard input with <)
- Some commands can accept data redirected to STDIN from a file:

```
$ tr 'a-z' 'A-Z' < .bash_profile
```

Shell – echo

```
$ echo "Hello, world"
Hello, world
```

```
$ echo Hello, world
Hello, world
```

```
$ echo ‘Hello, world’
Hello, world
```

- backslash (\)
- double quotation marks (")
- single quotation marks (')

Infosys | Building Tomorrow's Enterprise

Shell – echo

```
$ MYVAR = "Hello, world‟


$ echo $MYVAR
Hello, world


$ echo "$MYVAR"
Hello, world


$ echo '$MYVAR'
$MYVAR
```

Shell – echo

```
$ echo \$MYVAR
$MYVAR

$ echo \'$MYVAR\'
'Hello, world'

$ echo "'$MYVAR'"
'Hello, world'

$ echo '"$MYVAR"'
"$MYVAR"

$ echo \"$MYVAR\"
"Hello, world"
```

Shell – echo – wildcard expansion

```
$ ls *

$ echo *

$ echo ls *

'*' is expanded by the shell

$ touch p1.c p2.c p3.c p4.c
$ find . -name *.c
[[what is the output you expect]]

See what is actually happening using
$echo find . -name *.c
```

Shell – echo – wildcard expansion

See what is actually happening using:

```
$ echo find . -name *.c
```

This happens because *.c has been expanded by the shell resulting in find actually receiving a command line like this:

```
$ find . -name p1.c p2.c p3.c p4.c
```

That command is of course not going to work. Instead of doing things this way, you should enclose the pattern in quotes the wildcard:

```
$ find . -name '*.c' -print
```

Shell – echo brace expansion

```
$ echo {one,two,red,blue}
one two red blue

$echo one two red blue
one two red blue


$ echo {one,two,red,blue}fish
onefish twofish redfish bluefish

$ echo fish{one,two,red,blue}
fishone fishtwo fishred fishblue

$ echo fi{one,two,red,blue}sh
fionesh fitwosh firedsh fibluesh
```

Note: There are no spaces inside the brackets or between the brackets and the adjoining strings.

Shell – echo brace expansion

If you include spaces, it breaks things:

```
$ echo {one, two, red, blue }fish
{one, two, red, blue }fish

$ echo "{one,two,red,blue} fish"
{one,two,red,blue} fish
```

Spaces can be used if they're enclosed in quotes outside the braces or within an item in the comma-separated list

```
$ echo {"one ","two ","red ","blue "}fish
one fish two fish red fish blue fish

$ echo {one,two,red,blue}" fish"
one fish two fish red fish blue fish
```

Shell – echo brace expansion

You also can nest braces:

```
$ echo {{1,2,3},1,2,3}
1 2 3 1 2 3

$ echo {{1,2,3}1,2,3}
11 21 31 2 3

$ echo “Hello”{1..1000}

$ echo {a..z}

$ echo {0..1000}{0..1000}

$ echo {1..100..2}
```

Shell – echo brace expansion

How can brace expansion be used ?

```
$ touch myfile.conf
$ cp myfile.conf{,.bak}

$ ls
myfile.conf  myfile.conf.bak

$echo cp myfile.conf{,.bak}
```

list the files that match more than one pattern, it is easiest to do:

```
$ ls -l /etc/{*.conf,*.local,*.rc}

$ mkdir 1000files
$ cd 1000files
$ echo touch {1..1000}.txt
$ touch {1..1000}.txt
```

Shell – Protecting Variables using Braces

Why use brace around variables (how to avoid potential problems)

```
$ VAR=ABC

$ echo "$VAR"

$ echo "$VAR abc"

$ echo "$VAR-abc"

$ echo "$VAR_abc"

$ echo "$VARabc"

$ echo "${VAR} abc"
$ echo "${VAR}-abc"
$ echo "${VAR}_abc"
$ echo "${VAR}abc"
```

Shell - comments “#”

```
$ # This is a comment and it does nothing

$ echo "Hello, world" # This text is not processed
Hello, world

$ echo # This will not output

$ echo ‘Hash (#) can be quoted'
Hash (#) can be quoted

$ echo "# Is also same even in double quotes"
# Is also same even in double quotes
```

Shell - arithmetic

```
$ echo $((10+40))
50

$ echo $((5*(3+3)))
30

$ ANUM=10
$ echo $ANUM
10

$ echo $(($ANUM-2))
8

$ ANUM = $(($ANUMr+5))
$ echo $ANUM
15

$ RESULT=$(($ANUM-10))
$ echo $RESULT
5
```

Shell - Argument list too long error!!!

```
$ cd /tmp/1Mfiles

$ ls | wc

$ ls * | wc
```

Shell - exit status

The $? variable represents the exit status of the previous command that was executed. Exit status is a numerical value returned by every command upon its completion.

As a best practice, any scripts that exit (error or normal) must terminate with proper exit status.

```
$ ls
$ echo $?

$ ls badfile
$ echo $?

$ grep root /etc/passwd
$ echo $?

$ grep toor /etc/passwd
$ echo $?
```

Shell – exit status across pipes

```
$ false | true | true | false | false
$ echo "${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]}
${PIPESTATUS[3]} ${PIPESTATUS[4]}"


$ ls | grep . | grep -v xxx
$ echo "${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]}”


$ ls /nofile | grep . | grep -v xxx
$ echo "${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]}”


$ ls / | grep . | grep xxx
$ echo "${PIPESTATUS[0]} ${PIPESTATUS[1]} ${PIPESTATUS[2]}”
```

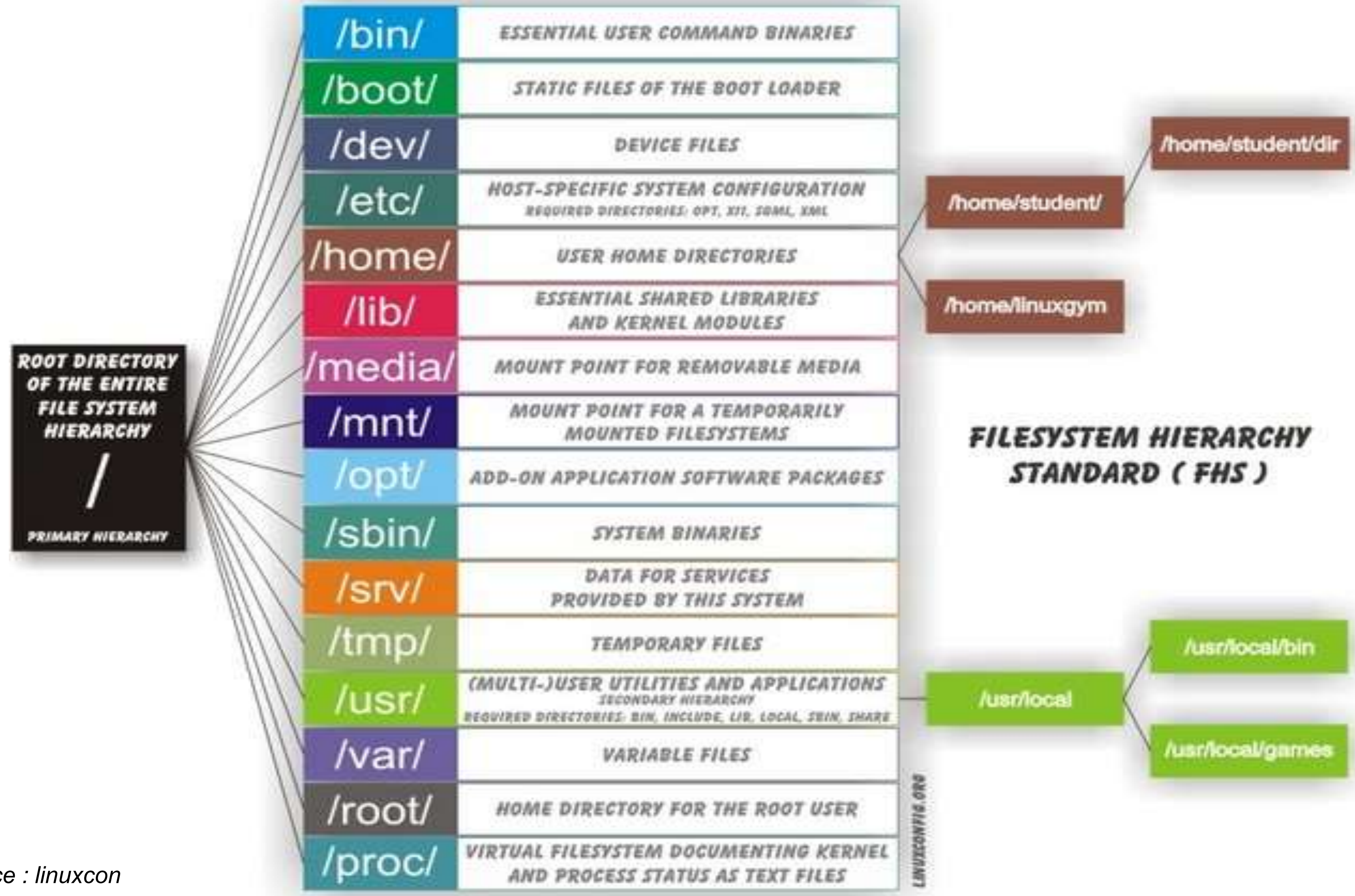
Filesystem Hierarchy

Mount	Short Description
/	The root directory, the top-level directory in the FHS. All other directories are sub directories of root, which is always mounted on some partition.
/bin	Essential command line utilities. Should not be mounted separately; otherwise, it could be difficult to get to these utilities when using a rescue disk.
/boot	Includes Linux start up files, including the Linux kernel. The default, 100MB, is usually sufficient for a typical modular kernel and additional kernels that you might install.
/dev	Hardware and software device drivers for everything from floppy drives to terminals. Do not mount this directory on a separate partition.
/etc	Most basic configuration files.
/home	Home directories for almost every user.
/lib	Program libraries for the kernel and various command line utilities. Do not mount this directory on a separate partition.
/media	The mount point for removable media, including floppy, DVDs.
/mnt	legacy mount point; formerly used for removable media.

Filesystem Hierarchy

Mount	Short Description
/opt	Common location for third-party application files.
/proc	Currently running kernel-related processes, including device assignments such as IRQ ports, I/O addresses, and DMA channels, as well as kernel configuration settings such as IP forwarding.
/root	The home directory of the root user.
/sbin	System administration commands. Don't mount this directory separately.
/tmp	Temporary files
/usr	Small programs accessible to all users. Includes many system administration commands and utilities
/var	Variable data, including log files and printer spools

Filesystem Hierarchy


ROOT DIRECTORY OF THE ENTIRE FILE SYSTEM HIERARCHY
/
PRIMARY HIERARCHY
/bin/ ESSENTIAL USER COMMAND BINARIES
/boot/ STATIC FILES OF THE BOOT LOADER
/dev/ DEVICE FILES
/etc/ HOST-SPECIFIC SYSTEM CONFIGURATION REQUIRED DIRECTORIES: OPT, X11, SGML, XML
/home/ USER HOME DIRECTORIES
/lib/ ESSENTIAL SHARED LIBRARIES AND KERNEL MODULES
/media/ MOUNT POINT FOR REMOVABLE MEDIA
/mnt/ MOUNT POINT FOR A TEMPORARILY MOUNTED FILESYSTEMS
/opt/ ADD-ON APPLICATION SOFTWARE PACKAGES
/sbin/ SYSTEM BINARIES
/srv/ DATA FOR SERVICES PROVIDED BY THIS SYSTEM
/tmp/ TEMPORARY FILES
/usr/ (MULTI-)USER UTILITIES AND APPLICATIONS SECONDARY HIERARCHY REQUIRED DIRECTORIES: BIN, INCLUDE, LIB, LOCAL, SBIN, SHARE
/var/ VARIABLE FILES
/root/ HOME DIRECTORY FOR THE ROOT USER
/proc/ VIRTUAL FILESYSTEM DOCUMENTING KERNEL AND PROCESS STATUS AS TEXT FILES
/home/student/
/home/student/dir
/home/linuxgym
FILESYSTEM HIERARCHY STANDARD (FHS)
/usr/local
/usr/local/bin
/usr/local/games
LINUXCONFIG.ORG

Source : linuxcon

Users/Groups and Security ….

- Every user is assigned a unique User ID number (UID); UID 0 identifies root
- Users are assigned to groups Each group is assigned a unique Group ID number (gid)
- Users cannot read, write or execute each others' files without permission
- All users in a group can share files that belong to the group
- Every file is owned by a UID and a GID
- Every process runs as a UID and one or more GIDs ; Usually determined by who runs the process
- If UID matches, user permissions apply Otherwise,
- If GID matches, group permissions apply
- If neither match, other permissions apply

Permissions….

```
$ ls -l calsum.sh
-rwxr-x--- 1 john staff 19080 Apr 1 18:26 calsum.sh
```

Read, Write and Execute for the owner, john
Read and Execute for members of the staff group
No access for all others

Changing File Ownership….

- Only root can change a file's owner
- Only root or the owner can change a file‘s group

- Ownership is changed with chown
- chown [-R] user_name file|directory

- Group-Ownership is changed with chgrp
- chgrp [-R] group_name file|directory

Tools of the trade…

grep

uname ps

cat

rm who

tac

awk

ls

du cp

mv

sed

tail split

mkdir

join

pwd head wc

more sort tr

vim

sum cut

wget

uniq

paste

man

find

tee

Tools of the trade – cat/tac

The cat program is a standard Unix utility that will output the contents of a specific file and can be used to concatenate and list files.

tac is a Linux command that allows you to see a file line-by-line backwards. It is named by analogy with cat.
Source : http://en.wikipedia.org/

```
$ cat file1
$ cat -n file1
$ cat -s file1
$ cat file1 file1
$ cat file1 file2 > big.file
$ tac file1
```

Don't use cat like below (wrong practice):

```
$ cat file.log | grep "something"
$ cat data | tr ' ' '\n'| wc
```

Can be rewritten as:

```
$ grep "something" file.log
$ tr ' ' '\n' < data | wc
```

Tools of the trade – head

Head prints the first N number of data of the given input. By default, it prints first 10 lines of each given file

```
$ head -n 5 /etc/passwd

$ head -5 /etc/passwd
```

Ignore last N lines

```
$ head -n -5 /etc/passwd
```

Can be used in pipe

```
$ find .  | head
```

Tools of the trade – tail

Tail prints the last N number of lines from given input. By default, it prints last 10 lines of each given file.

```
$ tail -n 5 /etc/passwd
```

```
$ tail -5 /etc/passwd
```

Ignore first N-1 lines

```
$ tail -n +5 /etc/passwd
```

View a growing log file

```
$ tail -f /var/log/messages
```

Again, it can be used in pipe

```
$ find .  | tail
```

Tools of the trade – head/tail - Print line between two range

Combine both command to print lines between two range

Start = 6
End = 10

```
$ head -10 /etc/passwd | tail -5
```

Or

```
$ tail -n +6 /etc/passwd | head -n 5
```

tail -n +6 : ignores lines upto the given line number (6)
head -n 5 : prints the first 5 line

TIP: Use cat –n to put line nos.

Tools of the trade – grep

Grep “something”

```
$ grep "something" mylogfile.log
```

Ignore case

```
$ grep -i "something" mylogfile.log
```

Everything Except the Search Term

```
$ grep -v "something" mylogfile.log
```

With line nos

```
$ grep -n "something" mylogfile.log
```

With occurrences

```
$grep -n "something" mylogfile.log
```

Tools of the trade – grep

Searching Multiple Files

```
$ grep "something" file1.log file2.log file3.log
$ grep "something" file*.log
```

Give only the Filenames

```
$ grep -l "something" file*.log
```

Printing lines before and after

```
$ grep -A 2 "something" mylogfile.log
$ grep -B 2 "something" mylogfile.log
$ grep -C 2 "something" mylogfile.log
```

Tools of the trade – grep - Line Anchors

Line Anchors
Use ^ for the beginning of the line
Use $ for the end of the line

```
$ grep ’root’ /etc/group
$ grep ’^root’ /etc/group
$ grep ’mount$’ /etc/fstab
```

Tools of the trade – egrep

egrep is same as ‘grep -E’ or ‘grep –extended-regex’

```
$ egrep "John|Joe" employee.txt

$ egrep "error|fail|timeout" myfile.txt

$ egrep [1,3] employee.txt

$ egrep [1-3] employee.txt
```

Tools of the trade – sort

Sort - sort/order lines in text files

```
$ sort somefile.txt

$ sort number.txt

$ sort -n number.txt

$ sort -r somefile.txt

$ sort -r number.txt

$ sort -r -n number.txt

$ sort -rn number.txt
```

Tools of the trade – sort

Sort - sort/order lines in text files

```
$ ps -ef | grep -v PID | sort -k2
```

Sort fields separated by ‘:’

```
$ sort /etc/passwd

$ sort -t ':' -k1 /etc/passwd

$ sort -t ':' -k3 /etc/passwd

$ sort -t ':' -n -k3 /etc/passwd
```

Tools of the trade – uniq

uniq - report or omit repeated lines

```
$ cat words.txt
and
and
single
who
who
who
notsingle
what
what
what
what
who
who
notsingle
and
and
and
and
```

Tools of the trade – uniq

uniq - report or omit repeated lines

```
$ uniq words.txt

$ uniq –i words.txt

$ uniq -c words.txt

$ uniq -d words.txt

$ uniq -u words.txt

$ uniq –u -c words.txt

$ uniq -D words.txt

$ uniq –c -d words.txt

$ uniq –c -D words.txt
```

Tools of the trade – uniq

uniq - report or omit repeated lines

```
$ sort words.txt | uniq

$ sort words.txt | uniq -i

$ sort words.txt | uniq -c

$ sort words.txt | uniq -d

$ sort words.txt | uniq -D

$ sort words.txt | uniq -c -d

$ sort words.txt | uniq -u

$ sort words.txt | uniq -c -u
```

Tools of the trade – tr

“tr” is used for translating, or deleting, or squeezing repeated characters

Simple

```
$ tr [:lower:] [:upper:]

$ tr a-z A-Z

$ tr [a-z,A-Z] [A-Z,a-z]

$ tr '{}' '()' < inputfile > outputfile
```

Tools of the trade – tr

More Interesting stuff…

#white-space to tabs

```
$ echo “Hello, this is cool" | tr [:space:] '\t'
```

#remove repetition of characters using -s

#note: there are multiple spaces in echo below

```
$ echo "Hello,   this   is    cool " | tr [:space:] '\t'
$ echo "Hello,   this   is    cool " | tr -s [:space:] '\t'
```

#Replace multiple continuous spaces with a single space

```
$ echo "Hello,   this   is    cool " | tr -s [:space:] ' '
```

Tools of the trade – tr

#Remove a specified characters using -d option
```
$ echo "Hello, this is cool stuff" | tr -d 't'

$ echo "Name: John, UserID: 735493" | tr -d [:digit:]
```

# Use Complement to reverse operation
```
$ echo "Name: John, UserID: 735493" | tr -cd [:digit:]
```

#Remove all special/non-printable characters
```
$ tr -cd [:print:] < access_log
```

#Remove newline and join all the lines
```
$ tr -s '\n' ' ' < William_Shakespeare.txt
```

#Give me word frequency….
```
$ tr -c a-zA-Z '\n' < William_Shakespeare.txt | grep -v "^$" |
sort | uniq -c | sort -n
```

Tools of the trade – find

The find utility recursively descends the directory hierarchy for each path seeking files that match the search expression. find detects infinite loops; that is, entering a previously visited directory that is an ancestor of the last file encountered

```
$ touch {p1,p2,p3}.c
$ touch {p1,p2,p3}.C

$ find .
$ find . -print
$ find . -name "*.c"
$ find . -iname "*.c"
```

Tools of the trade – find – Expressions (Logical Operators)

```
$ touch {p1,p2,p3}.cpp

$ find .  \( -name "*.c" -o -name "*.cpp" \)

$ find . -name '*.c' -o -name '*.cpp`

$ find .  -name "*.c*"

$ find . -name '*.c*' -a ! -name 'p1*`
```

Tools of the trade – find and Permissions

```
$ find . -perm 644 -type f
$ find . -perm 755 -type d

$ find . -perm 644 -type f -o  -perm 755 -type d

$ find . ! -perm 644 -type f
```

Matching a User's Permission Mode

```
$ find . -perm u=rwx,g=rx,o=rx
```

or

```
$ find . -perm a=rwx,g-w,o-w
```

Files with Write Access for other

```
$ find . -perm -o+w
```

Tools of the trade – find and size criteria

```
$ echo “01234567890123456789O” > SomeBytes.txt
$ find . –size +10  [[what is the output]]

$ find . –size -10  [[what is the output]]



$ find . -size +10000k -exec ls -sd {} \;

$ find . -size +10000k -printf "%h/%f,%s\n“

$ find . -size -10000k –type f -printf "%h/%f,%s\n“
```

Tools of the trade – find and size criteria

```
$ echo “01234567890123456789O” > SomeBytes.txt
$ find . –size +10  [[what is the output]]

$ find . –size -10  [[what is the output]]
```

By default it is in 512-byte blocks (i.e. this is the default if no suffix is used)

```
$ find . -size +10000k -exec ls -sd {} \;

$ find . -size +10000k -printf "%h/%f,%s\n“

$ find . -size -10000k –type f -printf "%h/%f,%s\n“
```

Tools of the trade – find and Time

-mtime : Mtime is the last modified time of a file

```
$ find . -mtime 7 -print
```

```
$ find . -mtime +6 -mtime -8 -print
```

-atime : last access

list all files that have not be read in thirty days or more

```
$ find . -type f -atime +30 -print
```

-ctime : This will have a more recent value if the owner, group, permission or number of links is changed, while the file itself does not.

Tools of the trade – find and newer (using comparisons)

```
$ touch -t MMDDHHMM.SS /tmp/reference


$ find . -newer /tmp/reference


$ find . -newer /tmp/reference
```

#Find files that are changing…

```
$ touch /tmp/testfile
$ find /var/log -newer /tmp/testfile -exec ls -lh {} \;
```

Tools of the trade – find and exec

```
$ find .  -name "*.c*" -exec /bin/ls {} \;

$ find .  -name "*.c*" -exec /bin/ls {} \; -exec /bin/md5sum {} \;

$ find .  -name "*.c*" -exec grep -l "Author: xxx" {} \;
```

Tools of the trade – find – common tasks

Putting ‘find’ to use: (what does these do?)

```
$ find . -type f -printf '%s %p\n'| sort -nr | head -10

$ find . -maxdepth 1 -printf '%s %p\n'|sort -nr|head

$ find . -type f -iname "*.mp4" -printf '%s %p\n'| sort -nr |
head -10

$ find . -type f -exec ls -al {} \; | sort -nr -k5 | head -n
10
```

Tools of the trade – cut

‘cut’ is used for text processing. Mainly used to extract portion of text from a file by selecting columns.

```
$ cat William_Shakespeare.txt
```

Select Column of Characters

```
$ cut -c2 test.txt
```

Select Column of Characters using Range

```
$ cut -c1-3 test.txt
```

Select Column of Characters using either Start or End Position

```
$ cut -c3- test.txt
```

Just extract 8 characters

```
$ cut -c-8 test.txt
```

Print everything

```
$ cut -c- test.txt
```

Tools of the trade – cut

Specific Field from a File

```
$ cut -d':' -f1 /etc/passwd
```

Multiple Fields from a File; home directory and shell

```
$ cut -d':' -f1,6 /etc/passwd
```

more fields

```
$ cut -d':' -f1-4,6,7 /etc/passwd
```

All Fields Except the Specified Fields

```
$ cut -d':' --complement -s -f7 /etc/passwd
```

Change Output Delimiter for Display

```
$ cut -d':'  -s -f1,6,7 --output-delimiter=',' /etc/passwd
```

Tools of the trade – awk

Awk Syntax:

```
awk '/search pattern1/ {Actions}
     /search pattern2/ {Actions}' file
```

In the above awk syntax:

```
search pattern is a regular expression.
Actions – statement(s) to be performed.
several patterns and actions are possible in Awk.
file – Input file.
Single quotes around program is to avoid shell not to
interpret any of its special characters.
```

Tools of the trade – awk

Awk Working Methodology

Awk reads the input files one line at a time.

- For each line, it matches with given pattern in the given order, if matches performs the corresponding action.
- If no pattern matches, no action will be performed.
- In the above syntax, either search pattern or action are optional, But not both.
- If the search pattern is not given, then Awk performs the given actions for each line of the input.
- If the action is not given, print all that lines that matches with the given patterns which is the default action.
- Empty braces with out any action does nothing. It wont perform default printing operation.
- Each statement in Actions should be delimited by semicolon.

source: http://www.thegeekstuff.com/

Tools of the trade – awk

```
## Create a file to test
$ head access_140731.log  > myfile.txt

#Print everything
awk '{print}' myfile.txt
[[equivalent to "cat myfile.txt"]]

#Print the lines which matches with the pattern.
$ awk '/ 404 /' myfile.txt
$ awk '/46.33.247.139/' myfile.txt

#Print only specific field
$ awk '{print $1, $10}' myfile.txt

#See what happens if we remove ","
$ awk '{print $1 $10}' myfile.txt
```

Tools of the trade – awk and NF

```
#Lets create a CSV output i.e. Put a ',' between the fields
$ awk '{print $1,",",$10}' myfile.txt


#Remove space between the fields
$ awk '{print $1","$10}' myfile.txt


#Another way to print entire line
$ awk '{print $0}' myfile.txt

#print how many fields are there in each input line
$ awk '{print NF}' myfile.txt


#print the value of the last filed NF
$ awk '{print $NF}' myfile.txt
```

Tools of the trade – awk and NF

```
#Now lets print the last three fields using NF-2, NF-1, NF
$ awk '{print $NF-2, $NF-1, $NF}' myfile.txt

What is the output you see? Is it as expected?
```

Tools of the trade – awk and NF

```
#Now lets print the last three fields using NF-2, NF-1, NF
$ awk '{print $NF-2, $NF-1, $NF}' myfile.txt
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "www.redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com"
-2 -1 "redlug.com‟
```

How do we fix this?

Tools of the trade – awk and NF

```
#Now lets print the last three fields using NF-2, NF-1, NF

$ awk '{print $(NF-2), $(NF-1), $NF}' myfile.txt
Chrome/33.0.1750.117 Safari/537.36" "redlug.com"
Gecko/20100101 Firefox/22.0" "redlug.com"
Safari/537.36 OPR/19.0.1326.59" "redlug.com"
Baiduspider/2.0; +http://www.baidu.com/search/spider.html)"
"www.redlug.com"
Chrome/33.0.1750.146 Safari/537.36" "redlug.com"
Version/5.0.6 Safari/533.22.3" "redlug.com"
Gecko/17.0 Firefox/17.0" "redlug.com"
Gecko/17.0 Firefox/17.0" "redlug.com"
Chrome/17.0.963.12 Safari/535.11" "redlug.com"
Chrome/17.0.963.12 Safari/535.11" "redlug.com‟

$ awk ’{ print "Record:", NR, "has", NF, "fields." }’
myfile.txt
```

Tools of the trade – awk – putting NF and NR to use

```
#Find the files that is less than 512 in the URL
$ awk '{print $10}' myfile.txt

$ awk '$10 < 512 ' myfile.txt
[[ by default prints complete line]]

$ awk '$10 < 512 {print $10}' myfile.txt

#What is the line number?
$ awk '$10 < 512 {print $NR, "====>", $10}' myfile.txt

#Print all “Chrome” users..
$ awk '$(NF-2) ~/Chrome/' myfile.txt

#What all versions
$ awk '$(NF-2) ~/Chrome/ {print $(NF-2)}' myfile.txt |sort –u

#What all versions and give count
$ awk '$(NF-2) ~/Chrome/ {print $(NF-2)}' myfile.txt|
sort|uniq -c
```

Tools of the trade – awk

```
#Put Header and Footer
$ awk 'BEGIN {print"=================================\n";
> print"    SUMMARY       \n";
> print"=================================\n"}
> {print $1,",",$10}
> END{print "----End of Report-----";}' myfile.txt
```

Tools of the trade – diff

diff analyzes two files and prints the lines that are different along with instructions for how to change one file in order to make it identical to the second file.

```
$ cat dlist1
Mercury
Earth
Mars
Jupiter
Saturn
Neptune

$ cat dlist2
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Neptune
```

Tools of the trade – diff

```
$ diff dlist1 dlist2
1a2
> Venus
```

After line 1 in the first file, a line needs to be added: line 2 from the second file." What needs to be added is shown.

```
$ diff dlist2 dlist1
2d1
< Venus
```

delete line 2 in the first file (`dlist2)` so that both files sync up at line 1.

Tools of the trade – diff

```
$ cat dlist1
Mercury
Earth
Mars
Jupiter
Saturn
Neptune

$ cat dlist2
Mercury
Earth
Jupiter
Mars
Saturn
Neptune
```

Tools of the trade – diff

```
$ diff dlist1 dlist2
3d2
< Mars
4a4
> Mars

$ diff dlist2 dlist1
3d2
< Jupiter
4a4
> Jupiter
```

Tools of the trade – diff

```
$ cat dlist1
1. Mercury
2. Earth
3. Mars
4. Jupiter
5. Saturn
6. Neptune

$ cat dlist2
1. Mercury
2. Earth
3. Mars
44. Jupiter
05. Saturn
6. Neptune
```

Tools of the trade – diff

```
$ diff dlist1 dlist2
4,5c4,5
< 4. Jupiter
< 5. Saturn
---
> 44. Jupiter
> 05. Saturn
```

4,5c4,5" means: "Lines 4 through 5 in the first file need to be changed in order to match lines 4 through 5 in the second file.

Tools of the trade – diff

diff output in contextual mode

```
$ diff -c dlist1 dlist2
*** dlist1      2015-02-07 16:21:06.020303800 +0530
--- dlist2      2015-02-07 16:21:33.482791100 +0530
***************
*** 1,6 ****
  1. Mercury
  2. Earth
  3. Mars
! 4. Jupiter
! 5. Saturn
  6. Neptune
--- 1,6 ----
  1. Mercury
  2. Earth
  3. Mars
! 44. Jupiter
! 05. Saturn
  6. Neptune
```

Tools of the trade – diff

diff output in Unified mode

```
$ diff -u dlist1 dlist2
--- dlist1      2015-02-07 16:21:06.020303800 +0530
+++ dlist2      2015-02-07 16:21:33.482791100 +0530
@@ -1,6 +1,6 @@
 1. Mercury
 2. Earth
 3. Mars
-4. Jupiter
-5. Saturn
+44. Jupiter
+05. Saturn
 6. Neptune
```

Tools of the trade – sed

sed is a “non-interactive” stream-oriented editor and since it is non-interactive, it can be used in scripts

Sed Working methodology

This is called as one execution cycle. Cycle continues till end of file/input is reached.

- Read a entire line from stdin/file.
- Removes any trailing newline.
- Places the line, in its pattern buffer.
- Modify the pattern buffer according to the supplied commands.
- Print the pattern buffer to stdout.

http://www.thegeekstuff.com/

Tools of the trade – sed

```
$ sed  '2'p dlist1
1. Mercury
2. Earth
2. Earth
3. Mars
4. Jupiter
5. Saturn
6. Neptune

$ sed -n '2'p dlist1
2. Earth

$ sed -n '2,4'p dlist1
2. Earth
3. Mars
4. Jupiter
```

Tools of the trade – sed

##- M~N - prints every Nth line starting from line M

```
$ sed -n '2~2'p dlist1
2. Earth
4. Jupiter
6. Neptune
```

##- ‘$’ Last Line

```
$ sed -n '$'p dlist1
6. Neptune
```

##- Match pattern

```
$ sed -n '/Mars/p' dlist1
3. Mars
```

Tools of the trade – sed

#Sed Substitution

```
$ echo "one Two two Two three Two four Two" | sed 's/Two/two/'
one two two Two three Two four Two

$ echo "one Two two Two three Two four Two" | sed
's/Two/two/g'
one two two two three two four two

$ echo "one Two two Two three Two four Two" | sed
's/two/two/i'
one two two Two three Two four Two

$ echo "one Two two Two three Two four Two" | sed
's/two/two/ig'
one two two two three two four two
```

Tools of the trade – sed

#Sed Substitution

```
$ echo "This is a bad line" | sed 's/bad/good/g'

#You can use other delimiter like @ # |
$ echo "This is a bad line" | sed 's@bad@good@g'

$ echo "/usr/local/bin:/usr/bin" |  sed
's//usr/local/bin//user/bin/g'

$ echo "/usr/local/bin:/usr/bin" |  sed
's/\/usr\/local\/bin/\/user\/bin/g'

$ echo "/usr/local/bin:/usr/bin" |  sed
's#/usr/local/bin#/user/bin#g'
```

Tools of the trade – sed

```
$ cat lines
01 This line may have one
10 This line may have one
3 This line may have one
5 This line may have one
21 This line may have one
22 This line may have one
33 This line may have one


$ sed 's/may have/has/' lines
01 This line has one
10 This line has one
3 This line has one
5 This line has one
21 This line has one
22 This line has one
33 This line has one
```

Tools of the trade – sed

```
$ sed '/1/s/may have/has/' lines
01 This line has one
10 This line has one
3 This line may have one
5 This line may have one
21 This line has one
22 This line may have one
33 This line may have one

$ sed '/1/!s/may have/does not have/' lines
01 This line may have one
10 This line may have one
3 This line does not have one
5 This line does not have one
21 This line may have one
22 This line does not have one
33 This line does not have one
```

Tools of the trade – sed

```
$ sed '/1/s/may have/has/' lines | sed '/1/!s/may have/does
not have/'
01 This line has one
10 This line has one
3 This line does not have one
5 This line does not have one
21 This line has one
22 This line does not have one
33 This line does not have one

# A better way…
$ sed '/1/s/may have/has/;/1/!s/may have/does not have/' lines
01 This line has one
10 This line has one
3 This line does not have one
5 This line does not have one
21 This line has one
22 This line does not have one
33 This line does not have one
```

Tools of the trade – sed

```
$ cat lines
This is line one
# comment 1
This is line two
# two blank lines below


This is line three
# comment and blank line below

This is line four
```

Tools of the trade – sed

```
$ sed 's/#.*//'  lines
This is line one

This is line two


This is line three


This is line four

$ sed '/^$/d'  lines
This is line one
# comment 1
This is line two
# two blank lines below
This is line three
# comment and blank line below
This is line four
```

Tools of the trade – sed

```
$ sed 's/#.*//'  lines | sed '/^$/d'
This is line one
This is line two
This is line three
This is line four

#Do it in better way
$ sed -e 's/#.*//;/^$/d' lines
This is line one
This is line two
This is line three
This is line four
```

Tools of the trade – df

df - report file system disk space usage

```
$ df

$ df -k

$ df -h

$ df  .

$ df -i

$ df -k | grep -v Filesystem |  sort -n -k4

$ df -h | grep -v Filesystem |  sort -h -r -k4
```

Tools of the trade – du

du - estimate file space usage

```
$ du

$ du .

$ du -s

$ du -s *

$ du -sk *

$ du -sh *

$ du -sk * | sort -n

$ du -sh * | sort -h
```

Tools of the trade – alias/unalias

alias - Define or display aliases
unalias - Remove the aliase

# display currently define aliases

```
$ alias
```

#Create a nee alias

```
$ alias ff='find . -name "*.c"'
$ alias
$ ff

$ alias fbig10="find . -type f -exec ls -s {} \; | sort -n -r
| head"

$ alias pstom="ps -aux ¦ grep tomcat"

$ alias ll='ls -l'
```

Tools of the trade – alias/unalias

#Alias can be used to override commands

```
$ cp source target
$ touch  source target
$ alias cp="cp -iv"      # interactive, verbose
$ cp source target
```

[[How does the above ‘cp’ alias work?]]

# to temporarily stop using 'cp' alias

```
$ \cp  source target
```

# to remove 'cp' alias

```
$ unalias cp
```

Tools of the trade – cp/rm/mkdir/rmdir

#cp - copy files and directories

```
cp [OPTION]... SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY

$ touch file1 file2 file3 file4
$ mkdir dir1 dir2 dir3
$ cp file1 file1.back
$ cp file1 file1
$ cp file1 file2
$ cp -i file1 file2

$ cp file1 dir1/.
$ cp file2 dir2/.
$ cp file1 file2 file3 dir3/.
```

[[why are we using /. At the end?]]

```
$ cp file4 dir4
$ rm dir4
$ cp file4 dir4/.
```

[[what do you see now?]]

Tools of the trade – cp/rm/mkdir/rmdir

#Lets create dir1/sdir2/sdir3/sdir4

```
$ mkdir dir1
$ cd dir1
$ mkdir sdir2
.
.
```

Or use the cool “-p” option of mkdir

```
$ cd ../..
$ mkdir -p dir1/sdir2/sdir3/sdir4
```

# Now create some files in each directory

```
$ touch dir1/myprog{1,2,3,4,5}.c
$ touch dir1/sdir2/c_prog{1,2,3,4,5}.c
$ touch dir1/sdir2/sdir3/perl_prog{1,2,3,4,5}.c
$ touch dir1/sdir2/sdir3/sdir4/pprog{1,2,3,4,5}.c
$ find dir1
```

Tools of the trade – cp/rm/mkdir/rmdir

# Lets make a backup of this directory
```
$ mkdir backup
$ cp dir1 backup/.
$ cd backup
$ ls
$ cd ..
```
[[what happened?]]

# -v will show in verbose mode
```
$ cp -rv dir1 backup/.
```

copy “sdir3” using relative directory to bak1
```
$ mkdir bak1
$ cp -rpv dir1/sdir2/sdir3 bak1/.
```

copy “sdir3” using relative directory to bak2 (from bak2 directory)
```
$ mkdir bak2
$ cd bak2
$ cp -rpv ../dir1/sdir2/sdir3 .
```

Tools of the trade – cp/rm/mkdir/rmdir

```
# Lets try deleting file and directories..
# Try deleting a directory
$ rmdir bak2

[[ what is the result ]]

$ rm bak2/sdir3/sdir4/pprog1.c
$ rm bak2/sdir3/sdir4/pprog2.c
$ cd bak2/sdir3/sdir4/
$ ls
$ rm *.c
$ cd ..
$ rmdir sdir4
.
.
$ rm -r bak2
```

Tools of the trade – cp/rm/mkdir/rmdir

# Lets see the times on the files in backup/*

```
$ find dir1 -ls | head -5
$ find backup/ -ls | head -5
```

[[what do you see?]]

```
$ cd backup
$ rm -r dir1
$ cd ..
```

use ‘-p’ to preserve the time

```
$ cp -rvp dir1 backup/.
$ find dir1 -ls | head -5
$ find backup/ -ls | head -5
```

Tools of the trade – cp/rm/mkdir/rmdir

```
$ mkdir adir
$ cd adir
$ touch newfile1.txt
$ chmod 000 newfile1.txt
$ cd ..
$ rm -r adir
```

[[what is the result]]

Tools of the trade – rm –rf

```
$ mkdir adir
$ cd adir
$ touch newfile1.txt
$ chmod 000 newfile1.txt
$ cd ..
$ rm -r adir
```

[[what is the result]]

$ rm -rf adir

Tools of the trade – xargs

xargs - Its a very powerful command used in along with find and grep to divide a big list of arguments into small list received from standard input.

Output with and without xargs

What does Xargs do by default : The default command executed by xargs is /bin/echo

```
$ echo Hello World | xargs echo
```

OR

```
$ echo Hello World | xargs
```

```
$ find /etc
```

v/s

```
$ find /etc | xargs
```

Tools of the trade – xargs

xargs and grep (very common)

```
$ find /etc -type f -exec grep "bash" {} \;
```
v/s
```
$ find /etc -type f | xargs grep "bash"
```

delete file using find and xargs

```
$ find dir1 -type f -name "*.c" -exec rm {} \;
```
v/s
```
$ find dir1 -type f -name "*.c" | xargs rm
```

Note: delete files with newlines or white space in file name with find and xargs
```
$ find dir1 -type f -name "*.c" -print0 | xargs -0 rm
```

Using -I {} "replace-str"
```
$ ls *.c | xargs -I {} echo mv {} {}_bkp
$ ls *.c | xargs -I {} mv {} {}_bkp
```

Tools of the trade – xargs

change multiline output into single line

```
$ ls *.txt
$ ls *.txt | xargs
```

Working on number of lines or files in each file using xargs

```
$ ls *.txt | xargs wc -l
$ find . -type -name "*.c" | xargs md5sum
```

Working with subset of arguments to xargs

```
$ echo 1 2 3 4 6 7 8 9 10 | xargs -n 3
$ find . -type -name "*.c" | xargs -n 2 echo
```

Cure for "Argument list too long"

```
$ gzip  *
```

""Argument list too long""

```
$ gzip * | xargs gzip
```

What is the difference between "find –exec" v/s "find along with xargs"

Tools of the trade – Archiving files - tar

tar - archiving utility

Creating an archive

```
$ tar cvf backup.tar dir1/
```

Test an archive

```
$ tar tvf backup.tar
```

Extract an archive

```
$ tar xvf backup.tar
```

Extract a single file from an archive

```
$ tar xvf backup.tar dir1/sdir2/sdir3/sdir4/pprog1.c
```

Extract multiple file from an archive

```
$ tar xvf backup.tar dir1/sdir2/sdir3/*.c
```

Tools of the trade – Archiving files - tar

Extract a single directory from an archive

```
$ tar xvf backup.tar dir1/sdir2/sdir3/sdir4/
```

Extract group of files from an archive

# Test it first

```
$ tar tvf backup.tar --wildcards '*2.c'
$ tar xvf backup.tar --wildcards '*2.c'
```

Tools of the trade – Compressing files – zip/unzip

compress single file using zip

```
$ zip file.zip file
```

Uncompress the files

```
$ unzip file.zip
```

Test a compressed file

```
$ unzip -t file.zip
```

compress multiple files

```
$ zip files.zip file1 file2 file3
```

compress all files in a directory into one file

```
$ zip -r dir1_backup.zip dir1
```

extract one file...

```
$ unzip  dir1_backup.zip  dir1/myprog5.c
```
[[what do you see]]

```
$ unzip -o dir1_backup.zip  dir1/myprog5.c
```

Tools of the trade – Compressing files – gzip/gunzip/bzip2/bunzip2

compress single file using gzip

```
$ touch gprog{1,2,3,4,5}.c
$ gzip  gprog1.c
$ gzip  -v gprog2.c
$ ls -l gprog*
```

[[what do you see? what happened to the original files?]]

#Uncompress the files

```
$ gunzip gprog1.c.gz
$ gunzip -v gprog2.c.gz

$ gzip  -k gprog1.c
$ gzip  -kv gprog2.c
$ ls -l gprog*
```

Tools of the trade – Compressing files – gzip/gunzip/bzip2/bunzip2

# Try compressing again
```
$ gzip -kv gprog1.c
```
[[what do you see?]]

```
$ gzip -f -kv gprog1.c
```

# gzip multiple files...

```
$ gzip -kv gprog3.c gprog4.c gprog5.c

$ gunzip -f -v gprog*.gz
```

#Let's try the recursive...
```
$ find dir1/
$ gzip -vr dir1/
$ find dir1/
$ gunzip -vr dir1/
$ find dir1/
```

Tools of the trade – Compressing files – gzip/gunzip/bzip2/bunzip2

How to zip group of files to a gzip single compressed file?

```
$ echo Prog1 > gprog1.c
$ echo Prog2 > gprog2.c
$ echo Prog3 > gprog3.c
$ echo Prog4 > gprog4.c
$ echo Prog5 > gprog5.c
$ ls -l gprog*.c

$ cat gprog1.c gprog2.c gprog3.c gprog4.c gprog5.c | gzip >
allfiles.gz
$ rm gprog*.c
$ ls -l gprog*.c
$ guznip -v allfiles.gz
$ ls -l gprog*.c
```

[[what is the listing output]]

Tools of the trade – Compressing files – gzip/gunzip/bzip2/bunzip2

```
$ ls -l allfiles
$ cat allfiles
```

Here we should know one behavior of gzip. gzip don't know how to add files to a single compress file!

How do we do it the right way?

```
$ tar cf - gprog*c  | gzip > allfiles.tar.gz
$ gunzip allfiles.tar.gz
$ tar xf allfiles.tar.gz
```

## Easier and cleaner way

```
$ tar cvfz allfiles.tar.gz gprog*.c
```

## Using this sequence, we can archive a directory also

```
$ tar cvfz dir1_backup.tar.gz dir1/
$ tar tvfz dir1_backup.tar.gz
$ tar xvfz dir1_backup.tar.gz
```

System Commands and Tools – Network - ifconfig

ifconfig - configure a network interface

```
$ ifconfig -a

$ ifconfig eth0
```

System Commands and Tools – Network - ping/traceroute

ping - send ICMP ECHO_REQUEST to network hosts
traceroute - print the route packets trace to network host

```
$ ifconfig eth0

$ ping use_ip_of_eth0
$ traceroute use_ip_of_eth0


$ cat /etc/hosts

$ ping replace_with_one_ip

$ traceroute replace_with_one_ip
```

System Commands and Tools – Network - netstat

netstat - Print network connections, routing tables, interface statistics, etc..

```
$ netstat
$ netstat -n


$ netstat -rn
$ netstat -r

$ netstat -i
```

List All Ports (both listening and non listening ports)

```
$ netstat -an | grep 22
```

List all tcp ports

```
$ netstat -at
```

List all udp ports using

```
$ netstat -au
```

System Commands and Tools – Network - netstat

netstat - Print network connections, routing tables, interface statistics, etc..

List only listening

```
$ netstat -lt
$ netstat -lu
```

statistics for all ports

```
$ netstat -s
$ netstat -st
$ netstat -su
```

Display PID and program names in netstat
[you may not see details if you don't have permission]

```
$ netstat -pt
```

Infosys Building Tomorrow's Enterprise

System Commands and Tools - Network - nslookup

nslookup - query Internet name servers interactively

```
$ nslookup www.google.com

$ nslookup 74.125.227.210


$ nslookup www.google.com 8.8.8.8

$ nslookup -query=mx infosys.com

$  nslookup -debug infosys.com
```

System Commands and Tools - Managing Processes - ps

List Running Processes

```
$ ps -ef
```

Process based on the UID

```
$ ps -f -u user10
```

Process based on the command

```
$ sleep 60 &
$ sleep 60 &
$ sleep 60 &
$ ps -f -C sleep
```

processes based on PIDs

```
$ ps -ef
$ ps -f  -p 25009,7258,2426
```

processes based on PPIDs

```
$ ps -f --ppid XXXX
```

System Commands and Tools - Managing Processes - ps

Processes in a Hierarchy/Tree format

```
$ ps -e -o pid,args --forest
```

List all threads

```
$ ps -ef
$ ps -ef | wc
$ ps -ef -L | more
$ ps -ef -L | wc
```

[[can be combined with -C or other options to show threads for particular combination]]

Find process with maximum memory usage

```
$ ps aux --sort pmem
```

System Commands and Tools - Managing Processes - top

top command displays the processes in the order of CPU usage

```
$ top
```

[[type 'q' to quit]]

#When top is running press 'M' to sort by "Memory"
#To see more sort options press 'O'

#Display Selected User in Top

```
$ top -u user12
```

#Display Only Specific Process

```
$ top -p xxxx, yyyy
```

System Commands and Tools - Managing Processes - top

#By default 'top' will show average of all CPUs
#TO see details for individual CPUs, press '1' when top is running

#Highlight Running Processes using 'z' or 'b'
[[you may not see this sometimes][

# Run 'top' for 5 counts and exit
```
$ top -n 5
```

# Output 'top' details continous form for 5 counts
# useful for redirecting to text or in scriptsand
```
$ top -b -n 5
```

System Commands and Tools - Managing Processes - kill

kill - terminate a process

Most common Signals used with kill command:

```
Signal 15, TERM (default) - Terminate cleanly
Signal 9, KILL - Terminate immediately
Signal 1, HUP - Re-read configuration files
```

Signals are specified by name or number when used with kill

The default signal is SIGTERM (terminate the process)

System Commands and Tools - Managing Processes - kill

```
$ sleep 300 &
$ sleep 300 &
$ sleep 300 &

$ ps -ef | grep 24715

$ kill -TERM 24715
$ kill -KILL 24716
$ kill -TERM 24717
```

System Commands and Tools - Managing Processes - vmstat

vmstat command reports information about processes, memory, paging, block IO, traps, and cpu activity.

Field in vmstat output
procs
r: The number of processes waiting for run time.
b: The number of processes in uninterruptible sleep.

memory
swpd: the amount of virtual memory used.
free: the amount of idle memory.
buff: the amount of memory used as buffers.
cache: the amount of memory used as cache.

swap
si: Amount of memory swapped in from disk (/s).
so: Amount of memory swapped to disk (/s).

System Commands and Tools - Managing Processes - vmstat

cpu
us: Time spent running non-kernel code. (user time, including nice time)
sy: Time spent running kernel code. (system time)
id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
wa: Time spent waiting for IO. Prior to Linux 2.5.41, shown as zero.

```
$ vmstat
$ vmstat 3
$ vmstat 3 4

$ vmstat 1 | awk '{ print strftime("%Y-%m-%d %H:%M:%S"), $0;
fflush(); }'
```

Users, Groups and Permissions – who, w, whoami, last

who - show who is logged on

w - Show who is logged on and what they are doing.

whoami - print effective userid

last - show listing of last logged in users

UNIX Shell Scripts

My First shell script and how to run it!

```
$ cat script1.sh
#!/bin/bash
echo "Hello, World!!!"

$ bash script1.sh

$ chmod 755 script1.sh

$ ./script1.sh
```

Second script – calling commands in it

```
$ echo "The date/time is : `date`"



$ cat script2.sh
#!/bin/bash
echo "SCRIPT BEGINS"
echo "Hello $LOGNAME"
echo
echo "Todays date is: `date '+%m/%d/%y'`"
echo "and the current time is: `date '+%H:%M:%S%n'`"
echo "SCRIPT FINISHED!!"
```

Second script – execute it in debug mode

To run an entire script in debug mode, add -x after `#!/bin/bash`

```
$ cat script2.sh
#!/bin/bash -x
echo "SCRIPT BEGINS"
echo "Hello $LOGNAME"
echo
echo "Todays date is: `date '+%m/%d/%y'`"
echo "and the current time is: `date '+%H:%M:%S%n'`"
echo "SCRIPT FINISHED!!“
```

-x Displays the line after interpreting
-v Displays the line before interpreting

```
$ bash -xv script2.sh
```

The Shell Environment

The Shell Environment

```
$ env
```

echo $$ - process ID of the current process
echo $! - process ID of the last background job started
echo $? - Contains the exit status of the most recent foreground process

How the shell works… what happens when you run a command

mycd.sh

```
#!/bin/bash
pwd
cd /tmp
pwd
```

How the shell works… what happens when you run a command

Try this (myenv.sh):

```
#!/bin/bash
EXPORT MYVAR1="Hello1"
EXPORT MYVAR2="Hello2"
echo $MYVAR1
echo $MYVAR2
```

```
$echo $MYVAR1
```

```
$echo $MYVAR2
```

How the shell works… what happens when you run a command

How do we run the earlier two scripts mycd.sh and myenv.sh and get the results we want:

```
$.  ./mycd.sh

$.  ./myenv.sh
```

OR use built-in “source” command

```
$source   ./mycd.sh

$source   ./myenv.sh
```

Storing values…

```
$ NOUSERS=`who|wc -l`
$ RUNTIME=`date '+%Y%m%d_%H%M%S'`
$ echo "Total users : ${NOUSERS} at ${RUNTIME}"
```

Exercise:

```
Write a program to log the following:

•  Date and Time of start of script [Started at : date]
•  No of Users [ No. Of Users = 120]
•  No of Uniq users with count
[
user1=10
User1=20
]
•  Date and Time the script completed [Completed at : date]
```

Exercise:

Write a program to log vmstat every 5 seconds to a file; The output file should have r,b,si,so,free(id) values only. The file should be CSV format. The first column should have date-time in DD/MM/YYYY HH:MM:SS.

if Statement…

```
#Syntax
if command
then
    block of statements
fi



$ cat script3_if.sh
#!/bin/bash
# Script name: script3_if.sh
MYVAR=OK
if [ "${MYVAR}" == "OK" ]
then
    echo "Variable has : ${MYVAR}"
fi
```

If/then/else Statement…

```
#Syntax
if command
then
    block of statements
else
    block of statements
fi

$ cat script4_if.sh
#!/bin/bash
# Script name: script4_if.sh
MYVAR=OK
if [ "${MYVAR}" == "OK" ]
then
    echo "Variable MYVAR in if-block : ${MYVAR}"
else
    echo "Variable does not match in else-block : ${MYVAR}"
fi
```

If/then/else Statement…

```
$ cat script5_if.sh
#!/bin/bash
# Script name: script5_if.sh

echo "ARG1 = ${1}"
echo "ARG2 = ${2}"
echo "ARG3 = ${3}"
echo "ARG4 = ${4}"
echo "ARG5 = ${5}"
```

If/then/elif/else Statement…

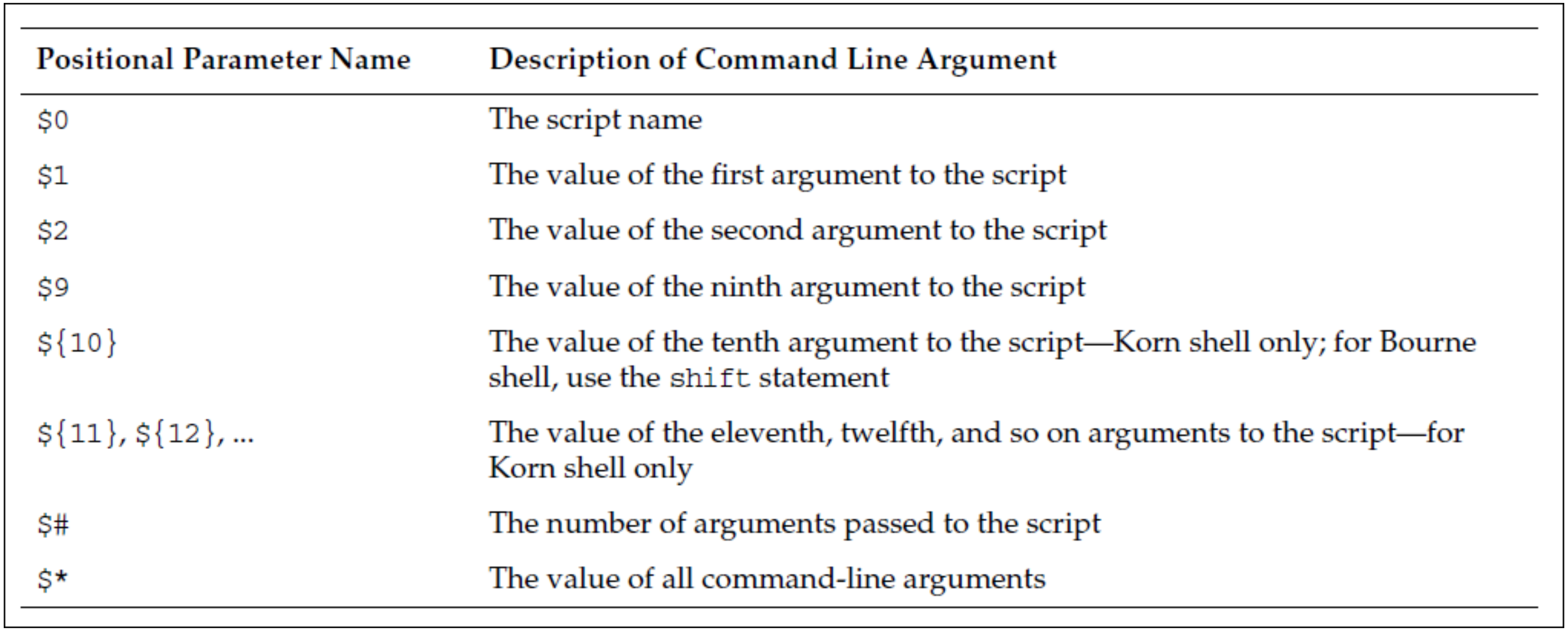
```
if/then/elif/else

if command1
then
    block of statements
elif command2
then
    block of statements
else
    block of statements
fi
```

If/then/elif/else Statement…

```
$ cat script6_if.sh
#!/bin/bash
# Script name: script6_if.sh
NAME=${1}
if [ "${NAME}" == "Binny" ]
then
    echo "First Name: ${NAME}"
elif [ "${NAME}" == "Raphael" ]
then
    echo "Last Name : ${NAME}"
else
    echo "Hello Stranger : ${NAME}"
fi
```

Script command line arguments



Positional Parameter Name	Description of Command Line Argument
`$0`	The script name
`$1`	The value of the first argument to the script
`$2`	The value of the second argument to the script
`$9`	The value of the ninth argument to the script
`${10}`	The value of the tenth argument to the script—Korn shell only; for Bourne shell, use the `shift` statement
`${11}, ${12}, …`	The value of the eleventh, twelfth, and so on arguments to the script—for Korn shell only
`$#`	The number of arguments passed to the script
`$*`	The value of all command-line arguments

Source: Bash Shell man pages

Script command line arguments

```
$ cat script7_args.sh
#!/bin/bash
# Script name: script7_args.sh

echo "Script Name:  ${0}"
echo "number of arguments passed:  ${#}"
echo "value of all command-line arguments:  ${*}"
echo "value of the first argument: ${1}"
echo "value of the fifth argument: ${5}"
echo "value of the tenth argument: ${10}"
echo "value of the 21st argument:  ${21}“

# Run with multiple args
# Run without any args
```

Script command line arguments

```
$ cat script8_if_check.sh
#!/bin/bash
# Script name: script8_if_check.sh
if [ $# -ne 1 ]
then
        print "USAGE: $0 enter_name "
        exit 1
fi
NAME=${1}
if [ "${NAME}" == "Binny" ]
then
        echo "First Name: ${NAME}"

elif [ "${NAME}" == "Raphael" ]
then
        echo "Last Name : ${NAME}"
else
        echo "Hello Stranger : ${NAME}"
fi
```

Bash Comparison Operators - Integer comparison

Integer comparison		
-eq	is equal to	`if [ "$a" -eq "$b" ]`
-ne	is not equal to	`if [ "$a" -ne "$b" ]`
-gt	is greater than	`if [ "$a" -gt "$b" ]`
-ge	is greater than or equal to	`if [ "$a" -ge "$b" ]`
-lt	is less than	`if [ "$a" -lt "$b" ]`
-le	is less than or equal to	`if [ "$a" -le "$b" ]`

Bash Comparison Operators - Integer comparison

String comparison		
`==`	is equal to	`if [ "$a" == "$b" ]`
`!=`	is not equal to	`if [ "$a" != "$b" ]`
`<`	is less than	`if [ "$a" \< "$b" ]`
`>`	is greater than	`if [ "$a" \> "$b" ]`
`-z`	string is null, that is, has zero length	`if [ -z "$a"]`

Bash file operators

File operators:

```
-a FILE   True if file exists.
-d FILE   True if file is a directory.
-f FILE   True if file exists and is a regular file.
-r FILE   True if file is readable by you.
-s FILE   True if file exists and is not empty.
-w FILE   True if the file is writable by you.
-x FILE   True if the file is executable by you.
```

Bash - Boolean AND, OR,and NOT

Boolean AND, OR,and NOT

```
AND operator is &&
OR operator is ||
NOT operator is !
```

Bash - read

```
$ cat script9_read.sh
#!/bin/bash
# Script name: script9_read.sh

echo "First Name :"
read FNAME

echo "Last Name"
read LNAME


echo "Hello ${FNAME} ${LNAME}"
```

Bash – IF - Integer comparison

```
$ cat script10_number.sh
#!/bin/bash
# Script name: script10_number.sh
echo "Please enter NUMBER1 number"
read NUMBER1
echo "Please enter NUMBER2 number"
read NUMBER2
if [ $NUMBER1 -eq 0 ] && [ $NUMBER2 -eq 0 ]
then
        echo "NUMBER1 and NUMBER2 are zero"
elif [ $NUMBER1 -eq $NUMBER2 ]
then
        echo "Both Values are equal"
elif [ $NUMBER1 -gt $NUMBER2 ]
then
        echo "$NUMBER1 is greater than $NUMBER2"
else
        echo "$NUMBER1 is lesser than $NUMBER2"
fi
```

Bash – IF – String comparison

```
$ cat script11_string.sh
#!/bin/bash
# Script name: script11_string.sh
echo "Please enter STRING1 :"
read STRING1
echo "Please enter STRING2 :"
read STRING2
if [ -z ${STRING1} ] || [ -z ${STRING2} ]
then
        echo "Either STRING1 or STRING2 is empty..."
elif [ ${STRING1} == ${STRING2} ]
then
        echo "Both Values are equal"
elif [ ${STRING1} > ${STRING2} ]
then
        echo "${STRING1} is greater than ${STRING2}"
else
        echo "${STRING1} is lesser than ${STRING2}"
fi
```

Bash – IF – file and directory…

```
$ cat script12_files.sh
#!/bin/bash
# Script name: script12_files.sh
read -p "Enter a file or directory path " FDNAME
if [ -z ${FDNAME} ]
then
        echo “No input passed..."
        exit 1
fi
if [ -d ${FDNAME} ]
then
    echo "${FDNAME} is a directory"
elif [ -f ${FDNAME} ]
then
    echo "${FDNAME} is a file"
else
    echo "${FDNAME} is not valid or it may be some other type of
file descriptor"
    exit 2
fi
```

Bash – IF – file and directory…

```
$ cat script12_files.sh
#!/bin/bash
# Script name: script12_files.sh
read -p "Enter a file or directory path " FDNAME
if [ -z ${FDNAME} ]
then
        echo "No input passed..."
        exit 1
fi
if [ -d ${FDNAME} ]
then
    echo "${FDNAME} is a directory"
elif [ -f ${FDNAME} ]
then
    echo "${FDNAME} is a file"
else
    echo "${FDNAME} is not valid or it may be some other type of
file descriptor"
    exit 2
fi
```

Exercise:

Write a program to monitor the disk utilization of a log file (e.g. /home/user10/lab/logs). The directory to be monitored and the output file should be configurable via variables. If its more than 100MB it should log the status to a file du_status.log in your directory with the following information:

1. Date and time in YYYY-MM-DD_HHMMSS
2. Current usage in KB

Exercise:

Write a program to monitor the disk utilization of a log file (e.g. /home/user10/lab/logs). The directory to be monitored and the output file should be configurable via variables. If its more than 100MB it should log the status to a file du_status.log in your directory with the following information:

1. Date and time in YYYY-MM-DD_HHMMSS
2. Current usage in KB

Problems: Files are overwritten and its in the same working directory

Apply fix:

1. Store it in a sub-directory (dulogs) in you PWD
2. If the file exists, append

Exercise:

```
Write a program to monitor the disk utilization of a log
file (e.g. /home/user10/lab/logs). The directory to be
monitored and the output file should be configurable via
variables. If its more than 100MB it should log the status
to a file du_status.log in your directory with the
following information:
1. Date and time in YYYY-MM-DD_HHMMSS
2. Current usage in KB

Apply fix $1:
1. Store it in a sub-directory (dulogs) in you PWD
2. If the file exists, append

Problems: One file is having all logs. Store it in
directory as YYYY/MM/DD and each day log name should be
"du_status_YYYYMMDD.log
```

Bash – case statement…

```
case value in
pattern1)
        statement1
        ...
        statementn
        ;;
pattern2)
        statement1
        ...
        statementn
        ;;
*)
        statement1
        ...
        statementn
        ;;
esac
```

Bash – case statement - example

```
$ cat script14_case.sh
#!/bin/bash
# Script name: script14_case.sh
echo -n "Do you want to print 'Hello World'? [yes or no]: "
read INPUT
case $INPUT in
        [yY]|[yY][Ee][Ss])
                echo "Hello World"
                exit 0
                ;;

        [nN]|[n|N][O|o])
                echo "I am not happy now... :-(";
                exit 1
                ;;
        *) echo "Please give proper input..."
           exit 2
            ;;
esac
```

Bash – case statement – start/stop/restart

```
$ cat script15_startup.sh
#!/bin/bash
# Script name: script15_startup.sh
case "$1" in
        'start')
                echo "Starting application"
                echo "This will call the START script"
                ;;
        'stop')
                echo "Stopping application"
                echo "This will call the STOP script"
                ;;
        'restart')
                echo "Restating application"
                echo "This will call the START script"
                echo "This will call the STOP script"
                ;;
        *) echo "Please give proper input..."
                echo "Usage: $0 [start|stop|restart]"

                ;;
esac
```

Bash – Here Document

```
The Here Document

$ cat script16_here.sh
#!/bin/bash
# Script name: script16_here.sh
echo "Select a name from below:"
cat <<EOF
    Sun
    Earth
    Moon
    Mars
    Jupiter
EOF
read -p "Enter your choice : " CHOICE

echo "You choice : $CHOICE‶

## EOF - must start in the first character position
```

Bash – Here Document - VARIABLES

```
$ cat script17_here.sh
#!/bin/bash
# Script name: script17_here.sh

SONGNAME="Twinkle, twinkle, little star"

cat > song.txt <<End-of-message
--------------------------------
Song : ${SONGNAME}
--------------------------------
Twinkle, twinkle, little star,
How I wonder what you are!
Up above the world so high,
Like a diamond in the sky.
--------------------------------
End-of-message
```

Bash – Here Document - strong-quote

```
$ cat script18_here_literal.sh
#!/bin/bash
# Script name: script18_here_literal.sh
PLANET1=Mars
PLANET2=Earth
PLANET3=Jupiter

cat <<End
-------------------------------------
with expansion
-------------------------------------
${PLANET1} - ${PLANET2} - ${PLANET3}
-------------------------------------
End

cat <<'End'
-------------------------------------
without expansion
-------------------------------------
${PLANET1} - ${PLANET2} - ${PLANET3}
-------------------------------------
End
```

Bash – Here Document – comment a section of script

```
$ cat script19_here_comment.sh
#!/bin/bash
# Script name: script19_here_comment.sh
PLANET1=Mars

:<<'COMMENTBLOCK'
echo "This will NOT echo when the script runs!!! "
# we are using "strong-quote"
PLANET2=Earth
PLANET3=Jupiter
COMMENTBLOCK

cat <<End
-------------------------------------
with expansion
-------------------------------------
${PLANET1} - ${PLANET2} - ${PLANET3}
-------------------------------------
End
```

Bash – For loop

The for loop allows you to specify a list of values. A group of statements is executed for each value in the list.

The for loop in the shell takes a list of words (strings) as an argument. The number of words in the list determines the number of times the statements in the for loop are executed.

The syntax for the for loop is:

```
for var in argument_list ...
do
    statement1
    ...
    statementn
done
```

Source: Shell Programming guide

Bash – For loop

```
#!/bin/bash
# Script name: xxxxx.sh

for PLANET in Mars Jupiter Earth Saturn Venus
do
    echo "Value of Planet is: ${PLANET}"
done
exit 0
```

Bash – For loop

```
$ cat xxxxx.sh
#!/bin/bash
# Script name: xxxxx.sh
read -p "Enter some text : " INPUT
for VAR in ${INPUT}
do
    echo "VAR now contains: $VAR"
done
exit
```

Bash – For loop

```
$ cat ex_for_arg.sh
#!/bin/bash
# Script name: ex_for_arg.sh

if [ $# -lt  1 ]
then
        echo "Please pass some arguments on command line"
        exit 1
fi

CINPUT=$*

for VAR in ${CINPUT}
do
        echo "VAR now contains: $VAR"
done
```

Bash – For loop

```
$ cat for_ls.sh
#!/bin/bash
# Script name: for_ls.sh

NUM=1

#call an external command
for VAR in `ls`
do
        echo "The file ${NUM} is  : $VAR"
        NUM=$((NUM+1))
done

exit 0
```

Exercise:

Write a script that

```
1. prints its name
2. checks if any parameter is passed
3. if so then, for every parameter, if its a directory,
list the 'inode' for all files in it OR if its a file its
inode directly
4. if no parameter, then asks for an input and does #3
```

Bash – While loop

The while Loop
The while loop allows you to repeatedly execute a group of statements while a command executes successfully. The syntax for the while loop is:

```
while command_control
do
    statement1
    ...
    statementn
Done
```

Source: Shell Programming guide

Bash – While loop

```
$ cat script_while.sh
#!/bin/bash
# Script name: script_while.sh


num=5
while [ $num -le 10 ]
do
    echo $num
    num=$((num+1))
done
```

Bash – While loop

```
$ cat script_while_read.sh
#!/bin/bash
# Script name: script_while_read.sh

NUM=1

# set the Internal Field Separator to a colon
IFS=:
while read UNAME f2 f3 f4 f5 UHOME USHELL
do
    echo  " ${NUM},${UNAME},${USHELL}"
    NUM=$((NUM+1))
done < /etc/passwd
```

Bash – break statement

The break Statement

The break statement allows you to exit the current loop. It is often used in an if statement that is contained within a while loop, with the condition in the while loop always evaluating to true. This is useful if the number of times the loop is executed depends on input from the user and not some predetermined number.

Source: Shell Programming guide

Bash – break statement

```
#!/bin/bash
# Script name: script_break.ksh
while true
do
        echo -n "Enter any number (0 to exit): "
        read NUM
        if [  ${NUM} -eq 0  ]
        then
                break
        else
                echo  "Square of $NUM is $(( NUM * NUM )). "
        fi
done
echo "script has ended"
```

Infosys Building Tomorrow's Enterprise

Bash – continue statement

The continue Statement

Use the continue statement within a loop to force the shell to skip the statements in the loop that occur below the continue statement and return to the top of the loop for the next iteration.

When you use the continue statement in a for loop, the variable takes on the value of the next element in the list. When you use the continue statement in a while, execution resumes with the test of the control_command at the top of the loop.

Source: Shell Programming guide

Bash – continue statement

```
#!/bin/bash
# Script name: script_while.sh
num=1
echo "Starting with : " $num
while [ $num -le 10 ]
do
        read -p "Do you want to skip printing the next number ?
(y/n) : " YESNO

        if [ ${YESNO} = "y" ]
        then
           echo "I am skipping a number!!"
           num=$((num+1))
           continue
        else
           echo $num
           num=$((num+1))
        fi
done
echo "All done!!!"
```

Bash – function

Functions are a way to group several UNIX / Linux commands for later execution using a single name for the group. Function accepts arguments.

```
Syntax to create a bash function:

function nameoffunction()
{
commands
.
.
}
```

Bash – function

```
$ cat script_function.sh
#!/bin/bash
# Script name: script_function.sh

function print_header_double()
{
        echo
"=========================================================="
}

function print_header_single()
{
        echo "----------------------------------------------------
------"
}

print_header_double
echo "My Directory status"
print_header_double
ls -l
print_header_single
```

Bash – getopts Statement

The getopts Statement

The getopts statement processes arguments on the command line that are options.

The getopts statement is most often used as the condition in a while loop, and the case statement is usually the command used to specify the actions to be taken for the various options that can appear on the command line.

Source: Shell Programming guide

Bash – getopts

```
#!/bin/bash
## Initialize Variables
COW_NUMBER=
SID=
VERBOSE=0

while getopts "hc:s:v" OPTION
do
     case $OPTION in
         h)
             print_usage_and_exit
             ;;
         c)
             COUNT=$OPTARG
             ;;
         s)
             SID=$OPTARG
             ;;
         v)
             VERBOSE=1
             ;;
```

Bash – getopts

```
?)
                    print_usage_and_exit
                    ;;
        esac
done

if [[ -z ${COUNT} ]] || [[ -z ${SID} ]]
then
        print_usage_and_exit
fi

#-------------------------

# Do all other stuff based on the inputs.....
#-------------------------

exit 0

#-------------------------
```

Exercise:

Write a script that
1. that saves the name os ONLY the file in the CURRENT directory to a file (dirfile.list)
2. Make sure that this list does NOT contain the file name 'dirfile.list'
3. for every file in the list that has an uppercase, rename it to lower case (e.g Prog.C --> prog.c)
4. DO not overwrite files
5. use 'while loop ' for processing the records
6. use 'continue' to skip records
7. Keep a log of activity and files that was renamed (details of changed only is required) in the following way
8. Print the statistics
9. See next slide for how the output should look.

Exercise:

```
==============================================================
  Script started at : YYYYMMDD_HHMMSS
==============================================================
  YYYYMMDD_HHMMSS - File  "Prog.C"  renamed to  "prog.c"...
  .
  .
  .
  .

  Summary:
        Total Files        :  34
        Files Renamed      :  14
        Files not Changed : 20

  Script completed at : YYYYMMDD_HHMMSS

==============================================================
```

Introduction to vi Editor

Vi Editor

vi Editor

Vi has two modes insertion mode and command mode.

The editor begins in command mode, where the cursor movement and text deletion and pasting occur.

Insertion mode begins upon entering an insertion or change command. [ESC] returns the editor to command mode (where you can quit, for example by typing :q!).

Most commands execute as soon as you type them except for "colon" commands which execute when you press the return key.

```
Format of vi commands: [count][command]

Example : 1024aa[Esc
```

Vi Editor

Input commands (end with Esc)

a	Append after cursor
i	Insert before cursor
o	Open line below
O	Open line above
:r file	Insert file after current line

All the above commands will make vi in input mode press Esc to come back to command mode.

Vi Editor

Change commands (Input mode)

cw	Change word (Esc)
cc	Change line (Esc) - blanks line
c$	Change to end of line
rc	Replace character with c
R	Replace (Esc) - typeover
s	Substitute (Esc) - 1 char with string
S	Substitute (Esc) - Rest of line with text
.	Repeat last change
~	Toggle upper and lower case

Vi Editor

Deletion commands

`dd or ndd`	Delete n lines to general buffer
`dw`	Delete word to general buffer
`dnw`	Delete n words
`db`	Delete previous word
`D`	Delete to end of line
`x`	Delete character

Vi Editor

File management commands

`:w name`	Write edit buffer to file name
`:wq`	Write to file and quit
`:q!`	Quit without saving changes
`ZZ`	Same as :wq
`:sh`	Execute shell commands (<ctrl>d)

Vi Editor

Window motions

`<ctrl>d`	Scroll down (half a screen)
`<ctrl>u`	Scroll up (half a screen)
`<ctrl>f`	Page forward
`<ctrl>b`	Page backward
`/string`	Search forward
`?string`	Search backward
`<ctrl>l`	Redraw screen
`<ctrl>g`	Display current line number and file information
`n`	Repeat search
`N`	Repeat search reverse
`G`	Go to last line
`nG`	Go to line n
`:n`	Go to line n
`z<CR>`	Reposition window: cursor at top
`z.`	Reposition window: cursor in middle
`z-`	Reposition window: cursor at bottom

Vi Editor

Cursor motions

H	Upper left corner (home)
M	Middle line
L	Lower left corner
h	Back a character
j	Down a line
k	Up a line
^	Beginning of line
$	End of line
l	Forward a character
w	One word forward
b	Back one word
fc	Find c
;	Repeat find (find next c)

Vi Editor

Undo commands

```
u           Undo last change
U           return the last line which was modified to its
             original state (reverse all changes in last
             modified line)
:q!         Quit vi without writing
:e!         Re-edit a messed-up file
Ctrl-R:     Redo changes which were undone (undo the undos)
```

Vi Editor

Rearrangement commands

yy or Y	Yank (copy) line to general buffer
nyy	Yank n lines to buffer
yw	Yank word to general buffer
ndd	Delete n lines to buffer
p	Put general buffer after cursor
P	Put general buffer before cursor
J	Join lines
nJ	Joins the next n lines together; omitting n joins the beginning of the next line to the end of the current line

Vi Editor

Playing with multiple files

```
:n
:rew
```

Move text from file old to file new

```
vi oldfile.txt
10yy        yank 10 lines to buffer a
:e newfile.txt
p           put text from a after cursor
```

#Write it to newfile.txt

```
:m,nw newfile.txt  Write lines m to n in file newfile.txt
:m,nw >>file     Saves lines m through n to the end of file
```

Vi Editor

Regular expressions (search strings)

^	Matches beginning of line
$	Matches end of line
.	Matches any single character
*	Matches any previous character
.*	Matches any character

Vi Editor

Search and replace commands

```
:[address]s/old_text/new_text/gic

Address components:
.          Current line
n          Line number n
.+m        Current line plus m lines
$          Last line
/string/ A line that contains "string"
%          Entire file
[addr1],[addr2] Specifies a range
```

Example:

```
Remove last character
:%s/.$//
 ^M
```

Vi Editor

Change some vi Parameters

:set list	Show invisible characters
:set nolist	Don't show invisible characters
:set number	Show line numbers
:set nonumber	Don't show line numbers
:set autoindent	Indent after carriage return
:set noautoindent	Turn off autoindent
:set showmatch	Show matching sets of parentheses as they are typed
:set noshowmatch	Turn off showmatch
:set showmode	Display mode on last line of screen
:set noshowmode	Turn off showmode
:set ignorecase	Ignore case on searches
:set ic	Ignore case on searches
:set noignorecase	Turn off ignore case
:set noic	Turn off ignore case
:set all	Show values of all possible parameters

Thank You

Infosys | Building Tomorrow's Enterprise