**INTRODUCTION TO LINUX OPERATING SYSTEM**

*Linux is an implementation of UNIX.*

*The Linux operating system is written in the C programming language.*

**What Is an Operating System**

In simple terms, an operating system is a manager. It manages all the available resources on a computer. These resources can be the hard disk, a printer, or the monitor screen. Even memory is a resource that needs to be managed. Within an operating system are the management functions that determine who gets to read data from the hard disk, what file is going to be printed next, what characters appear on the screen, and how much memory a certain program gets.

Once upon a time, there was no such thing as an operating system. The computers of forty years ago ran one program at a time. The computer programmer would load the program he (they were almost universally male at that time) had written and run it. If there was a mistake that caused the program to stop sooner than expected, the programmer had to start over. Because there were many other people waiting for their turn to try their programs, it may have been several days before the first programmer got a chance to run his deck of cards through the machine again. Even if the program did run correctly, the programmer probably never got to work on the machine directly. The program (punched cards) was fed into the computer by an operator who then passed the printed output back to the programmer several hours later.

As technology advanced, many such programs, or jobs, were all loaded onto a single tape. This tape was then loaded and manipulated by another program, which was the ancestor of today's operating systems. This program would monitor the behavior of the running program and if it misbehaved (crashed), the monitor could then immediately load and run another. Such programs were called (logically) monitors.

In the 1960's, technology and operating system theory advanced to the point that many different programs could be held in memory at once. This was the concept of "multiprogramming." If one program needed to wait for some external event such as the tape to rewind to the right spot, another program could have access to the CPU. This improved performance dramatically and allowed the CPU to be busy almost 100 percent of the time.

By the end of the 1960's, something wonderful happened: UNIX was born. It began as a one-man project designed by Ken Thompson of Bell Labs and has grown to become the most widely used operating system. In the time since UNIX was first developed, it has gone through many different generations and even mutations. Some differ substantially from the original version, like BSD (Berkeley Software Distribution) UNIX or Linux. Others, still contain major portions that are based on the original source code. (A friend of mine described UNIX as the only operating system where you can throw the manual onto the keyboard and get a real command.)

Linux is an operating system like many others, such as DOS, VMS, OS/360, or CP/M. It performs many of the same tasks in very similar manners. It is the manager and administrator of all the system resources and facilities. Without it, nothing works. Despite this, most users can go on indefinitely without knowing even which operating system they are using, let alone the basics of how the operating system works.

For example, if you own a car, you don't really need to know the details of the internal combustion engine to understand that this is what makes the car move forward. You don't need to know the principles of hydraulics to understand what isn't happening when pressing the brake pedal has no effect. an operating system is like that. You can work productively for years without even knowing what operating system you're running on, let alone how it works. Sometimes things go wrong. In many companies, you are given a number to call when problems arise, you report what happened, and it is dealt with. because of advances in both hardware design and performance, computers are able to process increasingly larger amounts of information. The speed at which computer transactions occur is often talked about in terms of billionths of a second. Because of this speed, today's computers can give the appearance of doing many things simultaneously by actually switching back and forth between each task extremely fast. This is the concept of multitasking. That is, the computer is working on multiple tasks "at the same time."

Another function of the operating system is to keep track of what each program is doing. That is, the operating system needs to keep track of whose program, or task, is currently writing its file to the printer or which program needs to read a certain spot on the hard disk, etc. This is the concept of multi-users, as multiple users have access to the same resources. in subsequent sections, I will be referring to UNIX as an abstract entity. The concepts we will be discussing are the same for Linux and any other dialect. When necessary, I will specifically reference where Linux differs.

**Properties of Linux**

**Linux Pros**

A lot of the advantages of Linux are a consequence of Linux' origins, deeply rooted in UNIX, except for the first advantage, of course:

**Linux is free:**

As in free beer, they say. If you want to spend absolutely nothing, you don't even have to pay the price of a CD. Linux can be downloaded in its entirety from the Internet completely for free. No registration fees, no costs per user, free updates, and freely available source code in case you want to change the behavior of your system.

**Linux is portable to any hardware platform:**

A vendor who wants to sell a new type of computer and who doesn't know what kind of OS his new machine will run (say the CPU in your car or washing machine), can take a Linux kernel and make it work on his hardware, because documentation related to this activity is freely available.

**Linux was made to keep on running:**

As with UNIX, a Linux system expects to run without rebooting all the time. That is why a lot of tasks are being executed at night or scheduled automatically for other calm moments, resulting in higher availability during busier periods and a more balanced use of the hardware. This property allows for Linux to be applicable also in environments where people don't have the time or the possibility to control their systems night and day.

**Linux is secure and versatile:**

The security model used in Linux is based on the UNIX idea of security, which is known to be robust and of proven quality. But Linux is not only fit for use as a fort against enemy attacks from the Internet: it will adapt equally to other situations, utilizing the same high standards for security. Your development machine or control station will be as secure as your firewall.

**Linux is scalable:**

From a Palmtop with 2 MB of memory to a petabyte storage cluster with hundreds of nodes: add or remove the appropriate packages and Linux fits all. You don't need a supercomputer anymore, because you can use Linux to do big things using the building blocks provided with the system. If you want to do little things, such as making an operating system for an embedded processor or just recycling your old 486, Linux will do that as well.

**Linux Cons**

**There are far too many different distributions:**

“Quot capites, tot rationes”, as the Romans already said: the more people, the more opinions. At first glance, the amount of Linux distributions can be frightening, or ridiculous, depending on your point of view. But it also means that everyone will find what he or she needs. You don't need to be an expert to find a suitable release.

When asked, generally every Linux user will say that the best distribution is the specific version he is using. So which one should you choose? Don't worry too much about that: all releases contain more or less the same set of basic packages. On top of the basics, special third party software is added making, for example, TurboLinux more suitable for the small and medium enterprise, RedHat for servers and SuSE for workstations. However, the differences are likely to be very superficial. The best strategy is to test a couple of distributions; unfortunately not everybody has the time for this. Luckily, there is plenty of advice on the subject of choosing your Linux. A quick search on Google, using the keywords “choosing your distribution” brings up tens of links to good advise. The Installation HOWTO also discusses choosing your distribution.

**Linux is not very user friendly and confusing for beginners:**

It must be said that Linux, at least the core system, is less userfriendly to use than MS Windows and certainly more difficult than MacOS, but... In light of its popularity, considerable effort has been made to make Linux even easier to use, especially for new users. More information is being released daily, such as this guide, to help fill the gap for documentation available to users at all levels.

**Is an Open Source product trustworthy?**

How can something that is free also be reliable? Linux users have the choice whether to use Linux or not, which gives them an enormous advantage compared to users of proprietary software, who don't have that kind of freedom. After long periods of testing, most Linux users come to the conclusion that Linux is not only as good, but in many cases better and faster that the traditional solutions. If Linux were not trustworthy, it would have been long gone, never knowing the popularity it has now, with millions of users. Now users can influence their systems and share their remarks with the community, so the system gets better and better every day. It is a project that is never finished, that is true, but in an ever changing environment, Linux is also a project that continues to strive for perfection.

**Current application of Linux systems**

Today Linux has joined the desktop market. Linux developers concentrated on networking and services in the beginning, and office applications have been the last barrier to be taken down. We don't like to admit that Microsoft is ruling this market, so plenty of alternatives have been started over the last couple of years to make Linux an acceptable choice as a workstation, providing an easy user interface and MS compatible office applications like word processors, spreadsheets, presentations and the like.

On the server side, Linux is well-known as a stable and reliable platform, providing database and trading services for companies like Amazon, the well-known online bookshop, US Post Office, the German army and such. Especially Internet providers and Internet service providers have grown fond of Linux as firewall, proxy- and web server, and you will find a Linux box within reach of every UNIX system administrator who appreciates a comfortable management station. Clusters of Linux machines are used in the creation of movies such as "Titanic", "Shrek" and others. In post offices, they are the nerve centers that route mail and in large search engine, clusters are used to perform internet searches.These are only a few of the thousands of heavy-duty jobs that Linux is performing day-to-day across the world.

It is also worth to note that modern Linux not only runs on workstations, mid- and high-end servers, but also on "gadgets" like PDA's, mobiles, a shipload of embedded applications and even on experimental wristwatches. This makes Linux the only operating system in the world covering such a wide range of hardware.

**What is the kernel?**

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system. A kernel can be contrasted with a shell (such as bash, csh or ksh in Unix-like operating systems), which is the outermost part of an operating system and a program that interacts with user commands. The kernel itself does not interact directly with the user, but rather interacts with the shell and other programs as well as with the hardware devices on the system, including the processor (also called the central processing unit or CPU), memory and disk drives

**What is a file system?**

A file system (sometimes written filesystem) is the way in which files are named and where they are placed logically for storage and retrieval. The DOS, Windows, OS/2, Macintosh, and UNIX-based operating systems all have file systems in which files are placed somewhere in a hierarchical (tree) structure. A file is placed in a directory or subdirectory at the desired place in the tree structure.File systems specify conventions for naming files. These conventions include the maximum number of characters in a name, which characters can be used, and, in some systems, how long the file name suffix can be. A file system also includes a format for specifying the path to a file through the structure of directories.

To gain access to files on another device, you must first inform the operating system where in the directory tree you would like those files to appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under thus-and-such a directory". The directory given to the operating system is called the mount point - it might, for example, be /mnt. The /mnt directory exists on many Unix-like systems (as specified in the Filesystem Hierarchy Standard) and is intended specifically for use as a mount point for temporary media like floppy disks or CDs. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

**What is shell in linux?**

A shell is a program that provides the traditional, text-only user interface for Unix-like operating systems. Its primary function is to read commands that are typed into a console (i.e., an all-text display mode) or terminal window (an all-text window) in a GUI (graphical user interface) and then execute (i.e., run) them.The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the operating system (at the very core of which is the kernel).

**What is a boot loader?**

Most simply, a boot loader loads the operating system. When your machine loads its operating system, the BIOS reads the first 512 bytes of your bootable media (which is known as the master boot record, or MBR). You can store the boot record of only one operating system in a single MBR, so a problem becomes apparent when you require multiple operating systems. Hence the need for more flexible boot loaders.The master boot record itself holds two things -- either some of or all of the boot loader program and the partition table (which holds information regarding how the rest of the media is split up into partitions). When the BIOS loads, it looks for data stored in the first sector of the hard drive, the MBR; using the data stored in the MBR, the BIOS activates the boot loader.

**What is Grub?**

Grand Unified Bootloader (GRUB)” .A small software utility that loads and manages multiple operating systems (and their variants).Linspire and Freespire OS use GRUB

**What is a partition?**

A partition is a section of a hard disk. When you format a hard disk, you can usually choose the number of partitions you want. The computer will recognize each partition as a separate disk, and each will show up under "My Computer".

**GENERAL-PURPOSR UTILITIES**

**date**

Print or set the system date and time

**Syntax**

date [OPTION]... [+FORMAT]

or:

date [OPTION] [MMDDhhmm[[CC]YY][.ss]]

**Description:**

**date** displays the current date and time when invoked without arguments. Providing arguments will format the date and time in a user-defined way or set the date. Only the superuser may set the date.

**Options**

%H hour :24 hour(00..23)

%M minute (00..59)

%S second (00..60)

%X locale's time representation (%H:%M:%S)

%Y year (1970...)

## Examples

## date

## will display:

fri may 16 08:25:53 IST 1997

The command:

**date** "+DATE: %m/%d/%y%nTIME: %H:%M:%S"

will display:

DATE: 11/21/87

TIME: 13:36:16

**date** 061316271985

sets the date to ``June 13, 1985, 4:27 PM''.

**cal**

Prints calendar

## Syntax

cal [[*month*] *year*]

## Description

**cal** prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *year* can be between 1 and 9999. *month* is a decimal number between 1 and 12. The calendar produced is a Gregorian calendar.

**OPTIONS**

-1 Display single month output. (This is the default.)

-3 Display prev/current/next month output.

-y Display a calendar for the current year

## Examples

cal 9 1850

prints the calendar for September, 1850 on the screen as follows:

     September 1850

S M Tu  W Th  F  S   
 1 2   3 4 5 6 7

8 9

**who**

Print who is currently logged in

(Lists currently logged on users username, port, and when they logged in)

**Syntax:**

who [*options*] [*file*] [am i]

**Example:**

**who**

root console jul 10 10:34

balaji tty01 jul 10 14:05

naveen tty05 jul 10 13:15

First coloumn shows login names.The second coloumn shows the device names of their respective terminals.The third coloumn shows the date and time of logging in.

**whoami**

Print the current user id and name

Syntax

Who am i

Example

kumar tty01 jul 15 15:09

**tty**

Print file name of terminal on standard input, print the file name of the terminal connected to standard input. It prints `not a tty' if standard input is not a terminal.

**Syntax:**

**tty**

**Example:**

**tty**

/dev/tty01

**uname**

Print system information, print information about the machine and operating system it is run on. If no options are given, `uname' acts as if the `-s' option were given.

Syntax

uname [*options*]...

**Options**

-a (all)

Print all of the below information.

-p (processor)

Print the machine's processor type

-r (release)

Print the operating system release.

-s (sysname)

Print the operating system name.

-v (version)

Print the operating system version

**Example:**

uname -a

=> Linux hayley 1.0.4 #3 Thu May 12 18:06:34 1994 i486

**passwd**

Modify a user password.

**Syntax:**

**passwd** [*options*...]

**Options:**

-d, --delete delete the password for the named account (root only)

-l, --lock lock the named account (root only)

-u, --unlock unlock the named account (root only)

**lock**

Reserve a terminal

**Syntax:**

lock [**-t** *timeout*]

**-t** *timeout* The time limit (default 15 minutes) is changed to *timeout*

minutes.

**Example:**

**lock**

password: \*\*\*\*\*\*\*\*

Re-enter password:\*\*\*\*\*\*\*\*

terminal locked by balu 0 minutes ago

The system will remain locked for 30 minutes in this condition. You can unlock the terminal.

By reentering the same password before that time. Otherwise it will log you out.

lock -45

you can set the duration of the lock (not exceeding 60 minutes)

**echo**

Display message on screen, writes each given STRING to standard output, with a space between each and a newline after the last one.

**Syntax:**

**echo** [*options*]... [*string*]...

**Example**

**echo** hello world

hello world

**echo** “hello world”

hello world

**tput**

**reset** - initialize a terminal or query terminfo database

## Syntax:

## tput [-T*type*] *capname* [*parms* ... ]

**Examples**

**tput** **cup** **0** **0**

Send the sequence to move the cursor to row **0**, column **0** (the upper left corner of the screen, usually known as the "home" cursor position).

**tput** **clear**

Echo the clear-screen sequence for the current terminal.

**tput** **cols**

Print the number of columns for the current terminal.

**bold=`tput** **smso`** **offbold=`tput** **rmso`**

Set the shell variables **bold**, to begin stand-out mode sequence, and **offbold**, to end standout mode sequence, for the current terminal. This might be followed by a prompt: **echo** **"${bold}Please** **type in** **your** **name:** **${offbold}\c"**

**tput** **cup** **23** **4**

Send the sequence to move the cursor to row 23, column 4.

**tput** **cup**

Send the terminfo string for cursor-movement, with no parameters substituted.

**bc**

Calculator mode

**Syntax:**

**bc**

**Example**:

bc

12+8

20

x=5 ; y=5 ; z=5 ;

a = x+y+z

a

15

**script**

Make typescript of terminal session

Syntax

**script** [**-a**] [*file*]

## Description:

## script makes a typescript of everything printed on your terminal. It is useful for students who need a hardcopy record of an interactive session as proof of an assignment, as the typescript file can be printed out later.

If the argument *file* is given, **script** saves all dialogue in *file*. If no file name is given, the typescript is saved in the file *typescript*.

Option:

**-a** Append the output to *file* or *typescript*, retaining the prior contents.

The script ends when the forked shell exits (a *control-D* to exit the Bourne shell.

Certain interactive commands, such as [**vi**](http://www.hmug.org/man/1/vi.php), create garbage in the type script file. **script** works best with commands that do not manipulate the screen, the results are meant to emulate a hardcopy terminal.

**Example**

**script**

script started, file is typescript

………….

…………

………….

Exit

scipt done, file is typescript

by default file is typescript. You can specify your filename

**script** filename

To append activities to the existing file

**script** -a

**NAVIGATING THE FLE SYSTEMS**

**pwd**

Print Present Working Directory

**Syntax:**

**pwd**

**Example:**

**pwd**

/user/kumar/progs

**cd**

Change Directory - change the current working directory to a specific Folder.

Syntax

**cd**

**Examples**

move to the sybase folder

$ **cd** /usr/local/sybase

$ **pwd**

/usr/local/sybase

Change to another folder

$ **cd** /var/log

Quickly get back

$ **cd** –

move up one folder

$**cd** ..

$ **cd** (Back to your home folder)

**mkdir**

Create new folder(s), if they do not already exist.

**Syntax:**

**mkdir** folder...

**mkdir** "Name with spaces"

**Example:**

**mkdir** path

it will create a directory

You can create a number of directories with one mkdir as follows

**$ mkdir**  naveen balaji kumar

**rmdir**

Remove folder(s), if they are empty.

**Syntax:**

**rmdir** folder...

**Example:**

**rmdir** logprog

it removes directory.

**ls**

List information about FILEs, by default the current directory.

Syntax

ls [Options]... [File]...

Options

-1 list one file per line

-a shows all files including **.,..** and those begging with a dot

-x Display file names in multiple columns

-f to identify directories and executable files

-r reverse the sort order

-t sorts files by modifications time

-s sort by file size

**Examples**

**ls -al**

total 109

drwxr-xr-x 18 root root 4096 Jun 9 21:12 ./

drwxr-xr-x 18 root root 4096 Jun 9 21:12 ../

drwxr-xr-x 2 root root 4096 Jun 9 21:14 bin/

drwxr-xr-x 3 root root 1024 Jun 9 20:32 boot/

drwxr-xr-x 6 root root 36864 Jul 12 10:26 dev/

drwxr-xr-x 34 root root 4096 Jul 12 10:25 etc/

drwxr-xr-x 5 root root 4096 Jun 9 21:28 home/

drwxr-xr-x 4 root root 4096 Jun 9 21:18 lib/

drwxr-xr-x 2 root root 16384 Jun 9 21:01 lost+found/

list ALL subdirectories

**ls \***

**HANDLING ORDINARY FILES**

**cat**

Display the contents of a file (concatenate)

**Syntax:**

**cat** filename

**Example**

Display a file  
$ **cat** myfile

Concatenate two files:

$ **cat** file1 file2 >> file3.dat

Put the contents of a file into a variable  
$my\_variable=`cat $myfile.txt

**cp**

Copy one or more files to another location  
  
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

**Syntax:**

**cp** [options] Source... Directory

**Options**

-i Interactive Copying

-r Copying Directory Structures

**Example**

**cp** prog progrs (if the progrs file already exists ,it will simply overwritten without any message)

**cp** –i sig signal

**cp**: overwrite signal? Y

**cp** –r sig newsig (copies all files and sub-directories in sig to newsig)

**rm**

Remove files (delete/unlink)

**Syntax:**

**rm** [ option] file name...

**Option**

-i Interactive Deletion

**Example:**

**rm** prog ( it delets a file)

**rm** \* (it deletes all files and directories simply it returns the prompt)

for confermation before removing each file by Interactive Deletion as follows:

**rm** –I prog prog1

prog : ?y

prog : ?n

**mv**

Move or rename files or directories.

**Syntax:**

**mv** [*options*]... *Source* *Dest*

**mv** [*options*]... *Source*... *Directory*

If the last argument names an existing directory, `mv' moves each other given file into a file with the same name in that directory. Otherwise, if only two files are given, it renames the first as the second. It is an error if the last argument is not a directory and more than two files are given.

**Examples**

Rename the file apple as orange.doc:

**mv** apple orange.doc

Move orange.doc to the Documents folder:

**mv** orange.doc ~/Documents/orange.doc

**more**

Display output one screen at a time, [less](http://www.ss64.com/bash/less.html) provides *more* emulation and extensive enhancements.

**Syntax:**

more

**Example**

more progs (it displays progs contents one page at a time )

**wc**

Print byte, word, and line counts, count the number of bytes, whitespace-separated words, and newlines in each given FILE, or standard input if none are given or for a FILE of `-'.

**Syntax:**

**wc** [*options*]... [*file*]...

**Options**

-c

--bytes

--chars

Print only the byte counts.

-w

--words

Print only the word counts.

-l

--lines

Print only the newline counts.

**Example :**

**wc** file

3 20 103 file (first it will displays lines, words, characters and filename.)

**wc** file1 file2

8 40 280 file1

10 60 200 file2

18 100 480 total (total as bonus)

**split**

Split a file into fixed-size pieces, creates output files containing consecutive sections of *INPUT* (standard input if none is given or INPUT is `-')

**Syntax:**

split [*options*] [*INPUT* [*PREFIX*]]

## Description

**split** reads *file* and writes it in pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with aa appended, and so on lexicographically, up to zz (only ASCII letters are used, a maximum of 676 files). If no output *name* is given, x is the default.

If no input *file* is given, or if - is given instead, the standard input file is used.

**Example:**

**split** file1 (by default creates a group of files xaa, xab, xac……………….)

ls

xaa xab..

**split**  -45 file10 parts (creates files partsaa, partsab, partsac…, etc. files having 45 lines)

**cmp**

Compare two files, and if they differ, tells the first byte and line number where they differ.  
  
You can use the `cmp' command to show the offsets and line numbers where two files differ. `cmp' can also show all the characters that differ between the two files, side by side.

**Syntax:**

**cmp** *options*... *FromFile* [*ToFile*]

## Description

cmp compares two files (if *file1* or *file2* is -, the standard input is used). Under default options, cmp makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

cmp recognizes the following options:

|  |  |  |
| --- | --- | --- |
| -l |  | Print the byte number (decimal) and the differing bytes (octal) for each difference (byte numbering begins at 1 rather than 0). |
| -s |  | Print nothing for differing files; return codes only. |

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.

**Example**

**cmp** tnsname tns

tnsname tns differ: char 11, line 1

(two files are compared byte by byte and the first mismatch is 11th character of 1 line )

**cmp** –l file1 file2 (gives a detailed list of the byte number and the differing bytes in octal for each character that differs in both files.)

**comm**

Compare two sorted files line by line and write to standard output, the lines that are common, plus the lines that are unique.

**Syntax:**

comm [*options*]... *File1 File2*

**Options**

-1 suppress lines unique to file1

-2 suppress lines unique to file2

-3 suppress lines that appear in both files

A file name of `-' means standard input.

Before `comm' can be used, the input files must be sorted using the collating sequence specified by the `LC\_COLLATE' locale, with trailing newlines significant. If an input file ends in a non-newline character, a newline is silently appended. The `sort' command with no options always outputs a file that is suitable input to `comm'.

With no options, `comm' produces three column output. Column one contains lines unique to FILE1, column two contains lines unique to FILE2, and column three contains lines common to both files. Columns are separated by a single TAB character.

The options `-1', `-2', and `-3' suppress printing of the corresponding columns.  
Unlike some other comparison utilities, `comm' has an exit status that does not depend on the result of the comparison. Upon normal completion `comm' produces an exit code of zero. If there is an error it exits with nonzero status.

**Example:**

**comm** file1 file2

**diff**

Display the differences between two files, or each corresponding file in two directories.

Each set of differences is called a "diff" or "patch". For files that are identical, `diff' normally produces no output; for binary (non-text) files, `diff' normally reports only that they are different.

**Syntax:**

diff [*options*] *from-file to-file*

**Example**

diff file1 file2s

**tee**

Redirect output to multiple files, copies standard input to standard output and also to any files given as arguments. This is useful when you want not only to send some data down a pipe, but also to save a copy.

**Syntax:**

tee [*options*]... [*file*]...

**Options:**

`-a'

`--append'

Append standard input to the given files rather than overwriting

them.

`-i'

`--ignore-interrupts'

Ignore interrupt signals.

**Example:**

You can use tee to save the output of the who command in a file, and u can display it.

**who** | **tee** user.lst

kumar tty01 jul 18 10:30

balaji tty02 jul 18 11:18

naveen tty03 jul 18 14:21

**BASIC FILE ATTRIBUTES**

**chmod**

Change access permissions

**Syntax:**

**chmod** [Options]... MODE[,MODE]... File...

**chmod** [Options]... NUMERIC\_MODE File...

**Numeric mode:**

From one to four octal digits  
Any omitted digits are assumed to be leading zeros.   
  
The first digit = selects attributes for the set user ID (4) and set group ID (2) and save text image (1)S  
The second digit = permissions for the user who owns the file: read (4), write (2), and execute (1)  
The third digit = permissions for other users in the file's group: read (4), write (2), and execute (1)  
The fourth digit = permissions for other users NOT in the file's group: read (4), write (2), and execute (1)

The octal (0-7) value is calculated by adding up the values for each digit  
User (rwx) = 4+2+1 = **7**  
Group(rx) = 4+1 = **5**  
World (rx) = 4+1 = **5**  
chmode mode = 0755

**Examples**  
  
Allow read permission to everyone:  
**chmod** 444 *file*  
  
Make a file readable and writable by the group and others:  
**chmod** 066 *file*   
  
Allow everyone to read, write, and execute the file:   
**chmod** 777 *file*

Bottom of Form

**Symbolic Mode**

The format of a symbolic mode is `[ugoa...][[+-=][rwxXs­tugo...]...][,...]'.   
  
Multiple symbolic operations can be given, separated by commas.  
  
A combination of the letters `ugoa' controls which **users**' access to the file will be changed:

The user who owns it (u)  
Other users in the file's group (g)  
Other users not in the file's group (o)  
All users (a)

If none of these are given, the effect is as if `a' were given, but bits that are set in the umask are not affected.  
  
all users (a) is effectively   
user + group + others  
  
The operator '+' causes the permissions selected to be added to the existing permissions of each file; '-' causes them to be removed; and '=' causes them to be the only permissions that the file has.  
  
The letters 'rwxXstugo' select the new **permissions** for the affected users:

Read (r),   
Write (w),   
Execute (or access for directories) (x),   
Execute only if the file is a directory or already has execute permission for some user (X),   
Set user or group ID on execution (s),   
Save program text on swap device (t),   
The permissions that the user who owns the file currently has for it (u),   
The permissions that other users in the file's group have for it (g),   
Permissions that other users not in the file's group have for it (o).

**Examples**:

Deny execute permission to everyone:

**chmod** a-x *file*  
  
Allow read permission to everyone:

**chmod** a+r *file*  
  
Make a file readable and writable by the group and others:

**chmod** go+rw *file*  
  
Allow everyone to read, write, and execute the file and turn on the set group-ID:

**chmod** =rwx,g+s *file*

**head**

Output the first part of files, prints the first part (10 lines by default) of each FILE; it reads from standard input if no files are given or when given a FILE of `-'.

**Syntax:**

**head** [*options*]... [*file*]...

**head** -NUMBER [*options*]... [*file*]...

**Example:**

**head** file5 (displays 10 lines from a file5, by default)

**head** -5 file5 (displays first 5 lines of the file5)

**tail**

Output the last part of files, print the last part (10 lines by default) of each FILE;   
tail reads from standard input if no files are given or when given a FILE of `-'.

**Syntax:**

tail [options]... [file]...

tail -Number [options]... [file]...

**Example:**

tail -18 file (displays last 18 lines )

**cut**

Divide a file into several parts (columns)  
Writes to standard output selected parts of each line of each input file, or standard input if no files are given or for a file name of `-'.

**Syntax:**

**cut** [OPTION]... [FILE]...

**paste**

Merge lines of files, write to standard output lines consisting of sequentially corresponding lines of each given file, separated by a TAB character.

**Syntax:**

**paste** [*options*]... [*file*]...

**sort**

**sort** text files, sort, merge, or compare all the lines from the files given (or standard input.)

**Syntax:**

**sort** [*options*] [*file*...]

**Options:**

**sort** has three modes of operation:

**sort** (the default), Merge (-m), and Check(-c)

-c Check whether the given files are already sorted: if they are not all sorted, print an error message and exit with a status of 1.

-m Merge the given files by sorting them as a group. Each input file should already be individually sorted. It always works to sort instead of merge; merging is provided because it is faster, in the case where it works.

**Examples**

Sort in descending (reverse) numeric order.

sort -nr

**uniq**

Uniquify files, write the unique lines in the given `InputFile', or standard input if nothing is given or for an INPUT name of `-'.

**Syntax:**

**uniq** [*options*]... [*InputFile* [*OutputFile*]]

**Options**

-u

--unique

Print only unique lines.

**nl**

Number lines and write files, writes each FILE to standard output, with line numbers added to some or all of the lines.

**Syntax:**

**nl** [*options*]... [*File*]...

**tr**

Translate, squeeze, and/or delete characters

**Syntax:**

**tr** [*options*]... *Set1* [*Set2*]

**REGULAR EXPRESSIONS AND THE grep FAMILY**

**grep**

Search file(s) for specific text.

**Syntax**

**grep** <*options*> "*Search String*" [*filename*]

**grep** <*options*> [-e *pattern*] [*file*...]

**grep** <*options*> [-f *file*] [*file*...]

**Options:**

-A NUM

--after-context=NUM

(GNU Extension)

Print NUM lines of trailing context after matching lines.

-B NUM

--before-context=NUM

(GNU Extension)

Print NUM lines of leading context before matching lines.

-C NUM

--context=[NUM]

(GNU Extension)

Print NUM lines (default 2) of output context.

`-c'

`--count'

Suppress normal output; instead print a count of matching lines for each input file. With the `-v', `--invert-match' option, count non-matching lines.

`-e PATTERN'

`--regexp=PATTERN'

Use PATTERN as the pattern; useful to protect patterns beginning with a `-'.

--h

--no-filename

(GNU Extension)

Suppress the prefixing of filenames on output when multiple files are searched.

-i

`--ignore-case'

Ignore case distinctions in both the pattern and the input files.

`-l'

`--files-with-matches'

Suppress normal output; instead print the name of each input file from which output would normally have been printed. The scanning of every file will stop on the first match.

`-n'

`--line-number'

Prefix each line of output with the line number within its input file.

option and its `-s' option behaved like GNU `grep''s `-q' option.

Shell scripts intended to be portable to traditional `grep' should avoid both `-q' and `-s' and should redirect output to `/dev/null' instead.

`-v'

`--invert-match'

Invert the sense of matching, to select non-matching lines.

-V

--version

(GNU Extension)

Print the version number of `grep' to the standard output stream.

This version number should be included in all bug reports.

**egrep**

Search file(s) for lines that match an extended expression

**Syntax:**

**egrep** [ options ] 'PATTERN' files ...

egrep is the same as `grep [-E](http://www.ss64.com/bash/grep2.html)'

all other options are the same as [grep](http://www.ss64.com/bash/grep.html)

The PATTERN is a regexp. In typical usage, the regexp is quoted to prevent the shell from expanding any of the special characters as file name wildcards. Normally, `egrep' prints the lines that matched. If multiple file names are provided on the command line, each output line is preceded by the name of the file and a colon.

**fgrep**

Search input files for lines that match a given pattern.

**Syntax:**

fgrep <options> ...

fgrep is the same as `grep [-F](http://www.ss64.com/bash/grep2.html)'

all other OPTIONS are the same as [grep](http://www.ss64.com/bash/grep.html)

It's a popular fallacy that `fgrep' stands for fast-grep.  
In fact `fgrep' means fixed-string grep**,** it has no performance boost over `grep'

**THE PROCESS**

**ps**

Process status, information about processes running in memory. If you want a repetitive update of this status, use top.

**Syntax:**

Display Process info

**ps** *option(s)*

**Options**

-A Select all processes. Identical to **-e**.

-f does full listing

-a Select all processes except session leaders and processes not associated with a terminal.

u display user-oriented format

**Running Jobs in a background**

The & is a shell’s operator used to run a process in the background. Terminate the command line with an & the command will run in the background.

**Syntax:**

$ command &.

**kill**

Unix communicate with the process with events called signals.

Each signal is identified by a signal number and designed to perform a specific function.

**Syntax:**

$ **kill** – signal pid

this will send a signal to process and the default handler of the signal is execulted.

**nice**

Run a command with modified scheduling priority, print or modify the scheduling priority of a job.

**Syntax:**

**nice** [*Option*]... [*Command* [*Arg*]...]

3333333333333

**SHELL PROGRAMMING (BOURNESHELL)**

**What is the BourneShell?**

BourneShell is a high level programming language and a command line interpreter.

The command to invoke the BourneShell is:

Command Format: **sh** [-acefhiknrstuvx] [args]

A Shell script is an executable plain file that contains UNIX and shell commands. To execute the shell script type the name of the script at the prompt. A simple shell script called shell\_ex is shown in the following example. The output from the execution of the shell is also shown.

**Sample Session:**

$**cat** shell\_ex

echo "This is a very simple shell procedure "

echo "created with the basic echo command "

echo "and three other very basic commands "

echo

ps

echo

who

echo

ls

$**sh** shell\_ex

This is a very simple shell procedure

created with the very basic echo command

and three other very basic commands

PID TTY TIME COMMAND

10443 rt02120 0:01 sh

10427 rt02120 0:04 ksh

sgavlick rt021e0 Sep 7 13:26

teacher rt021b0 Sep 7 14:39

memo

class\_notes

$

**Making a Bourne Shell Script Executable**

A BourneShell script is an ordinary file that contains commands which can be executed in sequence by entering one command at the BourneShell prompt. In order for a script to be executed, it must first be executable. This is done with the chmod command.

**Sample Session:**

$**cat** shell\_ex

echo "This is a very simple shell procedure "

echo "created with the basic echo command "

echo "and three other very basic commands "

echo

ps

echo

who

echo

ls

$

If the ls -l shell\_ex command were entered, we would see the protections assigned to this file.

Sample Session:

$**ls** -l shell\_ex

-rw-r--r-- 1 teacher class 66 Sep 7 10:24 shell\_ex

$

The character in column one is the type of file.

- = ordinary (plain) disk file

d = directory

b = block special file

c = character special file

p = fifo file ("named pipe") special file

l = symbolic link

Notice that the script file in the previous sample session has the following file protections:

User - Read and Write

Group - Read

Other - Read

No execute permissions have been granted for user, group, or other.

If we try to execute this script by typing its name, the following would result.

**Sample Session:**

$shell\_ex

shell\_ex: execute permission denied

$

This error message would indicate that execute permission was denied. The BourneShell script could not be executed. To change the permissions for the BourneShell script, use the chmod command.

**Sample Session:**

$**chmod** 755 shell\_ex

$**ls -l** shell\_ex

-rwxr-xr-x 1 teacher class 66 Sep 7 10:26 shell\_ex

$

Now that the permissions have been changed to allow user, group, and others to execute the file, it will execute properly.

**Sample Session:**

$shell\_ex

This is a very simple shell procedure

created with the basic echo command

and three other very basic commands

PID TTY TIME COMMAND

10443 rt02120 0:01 sh

10427 rt02120 0:04 ksh

sgavlick rt021e0 Sep 7 13:26

teacher rt021b0 Sep 7 14:39

The protections will work as you expect. Execute permission for the user will allow you (the owner) to run the BourneShell script. Group permissions allow anyone in your group to execute the script, and other permission allows anyone on the system to execute the script.

**Tracing Mechanisms**

It is possible to have a trace made of the BourneShell script as it executes. This is invaluable for debugging purposes. All that is required is to give an option to the BourneShell. This is done by including an option on the call to "sh". The command to do this is:

Command Format: **sh** [-acefhiknrstuvx] [args]

The option to turn on tracing is -x. For an example, let's trace the execution of the simple script shell\_ex.

**Sample session:**

$**cat** shell\_ex

echo "This is a very simple shell procedure "

echo "created with the basic echo command "

echo "and three other very basic commands”

echo

ps

echo

who

echo

ls

$

Execute the BourneShell script using the -x option on the call to the shell. The following sample session shows how to do this and it shows the results of the trace.

**Sample session:**

$**sh** -x shell\_ex

+ echo This is a very simple shell procedure

This is a very simple shell procedure

+ echo created with the basic echo command

created with the basic echo command

+ echo and three other very basic commands

and three other very basic commands

+ echo

+ ps

PID TTY TIME COMMAND

10443 rt01120 0:01 sh

10427 rt02120 0:04 ksh

+ echo

+ who

sgavlick rt021e0 Sep 7 13:26

teacher rt02120 Sep 7 14:39

+ echo

+ ls

memo

class\_notes

$

**USER, SHELL, AND READ-ONLY SHELL VARIABLES**

The BourneShell has no true numeric variables. It uses string variables to represent numbers, as well as text. String variables are able to take on the value of a string of characters. There are three types of variables in the BourneShell. They are user variables, BourneShell variables, and Read-only BourneShell variables.

You can declare, initialize, read, and modify user variables from a BourneShell script or from the command line. The BourneShell itself declares and initializes shell variables, but you can read and modify them. The BourneShell also initializes the read-only shell variables, and you can read but not modify them.

**User Variables**

It is legal to assign any sequence of non-blank characters as the name of a variable. The sample session below creates a variable called person and initializes it with the string Richard.

It is important to note that you must NOT precede or follow the equal sign with a space or TAB character.

This sample session indicates that person does not represent the string Richard. The string person is echoed as person. The BourneShell will only do the substitution of the value of the variable when the name of the variable is preceded with a dollar sign ($).

**Sample Sesssion:**

$**echo** person

person

$**echo** $person

Richard

$

If you want to have imbedded spaces in a variable, it is necessary to quote the string.

**Sample Session:**

$person='Richard and Kathleen'

$echo $person

Richard and Kathleen

$

The echo utility copies its arguments to the standard output. The command echo $person displays the value of the variable person. It will not display $person because the BourneShell doesn't pass $person as an argument. The leading dollar sign ($) causes the BourneShell to substitute the value of the variable and then passes that value to the utility. The echo utility then displays the value of the variable, not its name, never knowing that you called it with a variable. The BourneShell passed the same command line as if you had typed in echo Richard and Kathleen. The BourneShell can be prevented from doing this substitution by entering one of the following:

**Sample Sessions:**

$**echo** $person

Richard and Kathleen

$

In this session the contents of the variable person are displayed. The BourneShell made the substitution because the variable name person is preceded by a dollar sign ($).

$**echo** \$person

$person

$

In the above example the variable person is preceded by a dollar sign ($) but the dollar sign has a backslash (\) ahead of it. The backslash has the effect of cancelling the special meaning of the character following the backslash. In this case, the special meaning of the dollar sign is ignored and the substitution is not done.

$**echo** '$person'

$person

$

The single quote marks (') causes the characters between the marks to be taken as literal. The shell makes no attempt to interpret the meanings of these characters. The shell passes these characters on with no substitution.

$**echo** "$person"

Richard and Kathleen

$

The double quote marks do not prevent the shell from making substitution; and the value of the variable will be displayed by the utility.

**Shell Variables**

The BourneShell declares and initializes variables that determine such things as your home directory, what directories the shell will look in when you give commands, how often to look for mail, your prompt, and many other things. We will look at several of these BourneShell variables and their functions. You can assign new values to these variables from the command line or from the execution of the .profile file in your home directory.

**HOME**

The first BourneShell variable that we will look at is the HOME variable. By default, the home directory is the current working directory after you login. The system administrator determines your home directory when you establish an account and places that information in the /etc/passwd file. When you login, the BourneShell gets that pathname and assigns it to the HOME variable.

When you enter a cd command with no argument, the utility takes the name of the directory from the HOME variable and makes it the current working directory. If you change the HOME variable to another directory pathname, the utility will make the new directory the current working directory.

**Sample Session:**

$echo $HOME

/user0/rharding

$cd

$pwd

/user0/rharding

$HOME=/user0/rharding/eng

$cd

$pwd

/user0/rharding/eng

$

This example shows how the value of the HOME variable affects the cd utility. The cd command will use the value of the HOME variable as the pathname for the current working directory.

**IFS**

This is the internal-field separator BourneShell variable. You can always use a space or tab to separate characters on the command line. When you assign the IFS variable to another character, you can also use this character as the field separator.

**Example:**

$num\_args a:b:c:d .

This example shows only one argument, namely a:b:c:d.

$**IFS**=:

$num\_args a:b:c:d .

This example now shows four different arguments; each being separated by the new IFS, (:).

**MAIL**

The MAIL variable contains the name of the file that the mail (and mailx) utilities use to store your mail. Usually, the absolute pathname of this file is /usr/mail/name, where name is your login name.

**Example:**

$MAIL=/usr/mail/rharding .

**MAILPATH**

This variable contains a list of filenames separated by colons. If set, the BourneShell will inform you when any of these files are modified (i.e. when new mail arrives). Normally, this variable is not set.

**MAILCHECK**

This variable specifies how often, in seconds, the BourneShell will check for new mail. The default is 600 seconds. If set to 0, it will check for new mail each time before it gives you a prompt.

**PATH**

This BourneShell variable will describe the directories that will be searched looking for the program that you want to execute. The BourneShell looks in several directories for a file that has the same name as the command that you entered. The PATH variable controls this search path. Normally, the first directory searched is the current working directory. If the program is not found, the search continues in the /bin and then the /usr/bin directory. Generally, these directories contain executable programs. If the program is not found in one of these directories, the BourneShell reports that the program can't be found (or executed).

The PATH variable lists the pathnames in the order in which the search will proceed. The pathnames are separated by a colon (:). If nothing (null string) precedes the colon, that indicates to start the search at the current working directory.

**Example:**

$PATH=:/user0/rharding/bin:/bin:/usr/bin .

$ .

This PATH variable indicates to start the search for the program at the current working directory, then look in the directory /user0/rharding/bin, then /bin, and finally /usr/bin.

If each user has a unique path specified, each user can execute a different program by giving the same command. The search for the program stops when it is satisfied; thus, you can use the same name for your own programs as the standard UNIX utilities. To do this, simply put your program in one of the first directories that the BourneShell searches.

**PS1**

This is the BourneShell prompt which lets you know that the shell is waiting for you to give it a command. The default BourneShell prompt is a dollar sign ($). The shell stores the prompt as a string variable in PS1. When you change the value of this variable, the appearance of the prompt will change. When you are working on several different machines, it might be useful to have the prompt be the name of the machine you are working on.

**Sample Session:**

$**pwd**

/user0/rharding

$**PS1**='domax0:’

domax0:

Notice that prompt is now domax0:

**PS2**

This variable is called the secondary prompt. If the command is not completed on one line and must be continued on the next line, the prompt for that continued line is PS2. The default is >. This prompt indicates that the BourneShell is expecting you to finish the previous command line.

**Sample Session:**

$**echo** 'demonstration of prompt string

>2'

demonstration of prompt string

2

$PS2='Continue? '

$echo 'demonstration of

Continue? prompt string 2'

demonstration of

prompt string 2

$

Notice how the secondary prompt was changed to "Continue? ".

**Read-Only User Variables**

The contents of the user variables and the shell variables can be modified by the user. It is possible to assign a new value to them. The new value can be assigned from the dollar ($) prompt or from inside a BourneShell script. Read-only variables are different. The value of read-only variables can not be changed.

The variable must be initialized to some value; and then, by entering the following command, it can be made read only.

Command format: readonly variable\_name

variable\_name = name of the variable to be made read only

**Sample Session:**

$person=Kathleen

$readonly person

$echo $person

Kathleen

$person=Richard

person: is read only

$

The readonly command given without any arguments will display a list of all the read-only variables.

**Sample Session:**

$person=Kathleen

$readonly person

$example=Richard

$readonly example

$readonly

readonly person

readonly example

$

**Read-Only Shell Variables**

The read-only shell variables are similar to the read-only user variables; except the value of these variables is assigned by the shell, and the user CANNOT modify them.

**Name of the Calling Program**

The shell will store the name of the command you used to call a program in the variable named $0.

It has the number zero because it appears before the first argument on the command line.

**Sample Session:**

$**cat** name\_ex

echo 'The name of the command used'

echo 'to execute this script was' $0

$name\_ex

The name of the command used to execute this script was name\_ex

$

**Arguments**

The BourneShell will store the first nine command line arguments in the variables named $1, $2, ..., $9. These variables appear in this section because you cannot change them using the equal sign. It is possible to modify them using the set command.

**Sample Session:**

$cat arg\_ex

echo 'The first five command line'

echo 'arguments are' $1 $2 $3 $4 $5

$arg\_ex Richard Kathleen Douglas

The first five command line

arguments are Richard Kathleen Douglas

$

The script arg\_ex will display the first five command-line arguments. The variables representing $4 and $5 have a null value.

The BourneShell variable $\* represents all of the command-line arguments as shown in the following example.

**Sample Session:**

$cat display\_all

echo $\*

$display\_all Richard Kathleen Douglas

Richard Kathleen Douglas

$

The BourneShell variable $# contains the number of arguments on the command line. This is a string variable that represents a decimal number. You can use the expr utility to perform calculations with that number and test to perform logical tests on it.

**Sample Session:**

$cat num\_args

echo 'This script was called with'

echo $# 'arguments'

$num\_args Richard Kathleen Douglas

This script was called with

3 arguments

$

**Shift**

The shift command promotes each of the command-line arguments. The second argument, represented by $2, is now the first argument, represented by $1. The third becomes the second and so on until the last argument becomes the next to last. You can access only the first nine command-line arguments (as $1 through $9). The shift command gives you access to the tenth, and the first becomes unavailable. There is no "unshift" command that will return the arguments that are no longer available.

**Sample Session:**

$**cat** demo\_shift

echo 'arg1='$1 ' arg2='$2 ' arg3='$3

shift

echo 'arg1='$1 ' arg2='$2 ' arg3='$3

shift

echo 'arg1='$1 ' arg2='$2 ' arg3='$3

shift

echo 'arg1='$1 ' arg2='$2 ' arg3='$3

shift

$demo\_shift Richard Kathleen Douglas

arg1=Richard arg2=Kathleen arg3=Douglas

arg1=Kathleen arg2=Douglas arg3=

arg1=Douglas arg2= arg3=

demo\_shift: cannot shift

$

The BourneShell will display an error message when the script executes a shift command after it has run out of variables.

**set**

The Set command will display a list of all the variables that are set when it has no arguments.

**Sample Session:**

$set

HOME=/user0/teacher

IFS=

LOGNAME=richard

MAIL=/usr/mail/richard

MAILCHECK=600

PATH=:/bin:/usr/bin

PS1=$

PS2=>

SHELL=/bin/sh

TERM=vt100

TZ=MST7MDT

$

When set is called with arguments, it sets the value of the command-line arguments ($1-$n) to the arguments. The example sets the first three arguments.

**Sample Session:**

$cat set\_ex

set who really cares

echo $#: $\*

$set\_ex

3: who really cares

$

**expr**

The expr command will perform arithmetic in the BourneShell.

Command format: expr expression

See Appendix C for a complete list of expressions

The arguments are taken as an expression. After the evaluation has taken place, the result is written to standard output. The terms of the expression must be separated by blanks. Special characters to the shell must be escaped. Strings containing blanks or other special characters must be quoted.

**Sample Session:**

$expr 7 + 8 + 10

25

$expr 10 - 8

2

$expr 10 '\*' 4

40

$expr 135 / 5

27

$

expr will also work with user defined variables as in the following

**Example:**

**Sample Session:**

$cat data

8

15

25

$cat express

count=0

tot=0

for a in `cat data`

do

tot=`expr $tot + $a`

count=`expr $count + 1`

done

avg=`expr $tot / $count`

echo "The average is $avg"

$

Let's execute the script "express" with tracing on so we can follow the execution.

**Sample Session:**

$sh -x express

count=0

tot=0

+ cat data

+ expr 0 + 8

tot=8

+ expr 0 + 1

count=1

+ expr 8 + 15

tot=23

+ expr 1 + 1

count=2

+ expr 23 + 25

tot=48

+ expr 2 + 1

count=3

+ expr 48 / 3

avg=16

+ echo The average is 16

The average is 16

$

**POSITIONAL PARAMETERS**

A BourneShell script can also read in command-line arguments. The first argument is referred to as $1, the second is $2, and so on. Command-line arguments are referred to as positional parameters.

Let's look at an example BourneShell script to see how these are used.

**Sample Session:**

$cat neat\_shell

echo $1 $2 $3

echo $0 is the name of the shell script

echo "There were $# arguments."

echo $\*

$

Insure that the BourneShell script is executable by issuing this command:

**Sample Session:**

$chmod a+x neat\_shell

$

Now, if we type the name of the BourneShell script with no arguments, we get the following results.

**Sample Session:**

$neat\_shell

neat\_shell is the name of the shell script

There were 0 arguments.

$

In this sample session, there were no arguments given so none were printed. $0 is the positional parameter that refers to the name of the script. Since there were no arguments given with this invocation of neat\_shell, there were zero arguments listed.

**Reading Input Into a Shell Variable**

The BourneShell script can read user input from standard input. The read command will read one line from standard input and assign the line to one or more variables. The following example shows how this works.

**Sample Session:**

$cat read\_script

echo "Please enter a string of your choice"

read a

echo $a

$

This simple script will read one line from standard input (keyboard) and assign it to the variable a.

**Sample Session:**

$read\_script

Please enter a string of your choice

Here it is

Here it is

$

The line read from standard input can also be assigned to several variables as shown in the following example.

**Sample Session:**

$cat reads

echo "Please enter three strings"

read a b c

echo $a $b $c

echo $c

echo $b

echo $a

$

This time, we will turn on the trace mechanism and follow the execution of this BourneShell script.

**Sample Session:**

$sh -x reads

+ echo Please enter three strings

Please enter three strings

+ read a b c

this is more than three strings

+ echo this is more than three strings

this is more than three strings

+ echo more than three strings

more than three strings

+ echo is

is

+ echo this

this

$

It is interesting to note that the spaces separate the values for the variables a,b, and c. For example, the variable a was assigned the string this, the variable b was assigned the string is, and the remainder of the line was assigned to c (including the spaces).

**Sample Session:**

$**cat** read\_ex

echo 'Enter line: \c'

read line

echo "The line was: $line"

$

In this example, the \c option will suppress the carriage return. The single quote marks protect the backslash from being interpreted by the shell. Also notice that the double quote marks have no effect on the substitution of the variable line.

**Sample Session:**

$read\_ex

Enter line: All's well that ends well

The line was: All's well that ends well

$

**Command Substitution**

You can execute a command by enclosing it within two grave accent marks [these are sometimes called backquotes (`)]. The BourneShell will replace the command and the grave marks with the output from the command.

**Sample Session:**

$cat dir

dir=`pwd`

echo 'You are using the' $dir 'directory'

$

**Sample Session:**

$dir

You are using the /user0/rharding directory

$

The important thing to notice here is that the pwd command was executed; and the output, /user0/rharding, was then assigned to the variable dir.

It is not necessary to assign the output of a command to a variable as shown in the previous example. The command substitution can occur directly as shown in the next example.

**Sample Session:**

$**cat** dir2

echo 'You are using the' `pwd` 'directory'

$dir2

You are using the /user0/rharding directory

$

One final example will show a practical use of command substitution. This BourneShell script will use the date command to provide the date in a useful format.

The normal output from the date command looks like the following.

**Sample Session:**

$date

Wed Sep 12 18:02:05 MDT 1990

$

Here's a BourneShell script that rearranges the output into a more useable format.

**Sample Session:**

$**cat** dateset

set `date`

echo $\*

echo

echo 'Argument 1:' $1

echo 'Argument 2:' $2

echo 'Argument 3:' $3

echo 'Argument 4:' $4

echo

echo $2 $3, $6

$dateset

Wed Sep 12 18:02:05 MDT 1990

Argument 1: Wed

Argument 2: Sep

Argument 3: 12

Argument 4: 18:02:05

Sep 12, 1990

$

The first command in the BourneShell script dateset uses the grave accent marks to set the command-line argument variables to the output of the date command. The next commands show the first four of these argument variables. The final command displays the arguments in a different order that could be useful in a report or a letter.

**Comments in BourneShell Scripts**

Comments can be inserted into the BourneShell script by beginning each comment line with the pound symbol (#) or a colon (:). All characters after the comment character will be ignored by the shell. The only exception to this rule is that the first character of the first line must not be a pound symbol; if the first character is a pound sign, the BourneShell tries to execute the script as if it was written in CShell syntax.

**Sample Session:**

$**cat** com\_sub

# The first line sets your present working directory

# to the variable 'directory'

directory=`pwd`

# The second line sets the date to the variable 'when'

when=`date`

: The third line will echo on the screen

echo "You are in $directory on $when"

: You could have said echo :

: "You are in `pwd` on `date`"

: to have a one line program

$

**BourneShell Environment - Exporting Variables**

Within a process, you can declare, initialize, read, and modify variables. The variable is local to that process. When a process forks a child process, the parent process does not automatically pass the value of the variable to the child process.

Here is an example of the variables not being exported.

**Sample Session:**

$**cat** no\_export

car=mercedes # set the variable

echo $0 $car $$ # $0 = name of file executed

# $car =value of variable car

# $$ = PID number (process id)

inner # execute another BourneShell script

echo $0 $car $$ # display same as above

$**cat** inner

echo $0 $car $$ # display variables for this process

$**chmod** a+x no\_export

$**chmod** a+x inner

$no\_export

no\_export mercedes 4790

inner 4792

no\_export mercedes 4790

$

When no\_export was executed, it, of course, assigned a value of mercedes to the variable car and printed it out. The call to inner created a child process. Its PID is 4792, while the parent PID is 4790. Notice, when inner tried to print the value of car, it printed nothing. The reason is because the value of car was not passed by the parent.

Can the value be passed from parent to child process? Yes, by using the export command. Let's look at an example.

**Sample Session:**

$**cat** export\_it

car=mercedes

export car

echo $0 $car $$

inner1

echo $0 $car $$

$**cat** inner1

echo $0 $car $$

car=chevy

echo $0 $car $$

$**chmod** a+x export\_it

$**chmod** a+x inner1

$export\_it

export\_it mercedes 4798

inner1 mercedes 4800

inner1 chevy 4800

export\_it mercedes 4798

$

In the export\_it BourneShell script, the variable car was initialized to mercedes; and then it was exported. This means that the value of car is now available to a child process. When inner1 prints out the value of car it has the value of mercedes. This is

as we expect because the value of car was exported from the parent.

The next line of inner1 changes the value of car to chevy. This is shown in the next line of the sample session. The last line of the session shows the return to the parent process and the value is still mercedes. How is this possible?

Exporting variables is only valid from the parent to the child process. The child process cannot change the parent's variable.

**CONTROL CONSTRUCTS**

The BourneShell control constructs can alter the flow of control within the script. The BourneShell provides simple two-way branch if statements and multiple-branch case statements, plus for, while, and until statements.

In discussing these control structures, the BourneShell keywords will be in bold type and the normal type are the user supplied items to cause the desired effect in command format boxes.

**Types of Tests Used with Control Constructs:**

The test utility evaluates expressions and returns a condition indicating whether or not the expression is true (equal to zero) or false (not equal to zero). There are no options with this utility. The format for this utility is as follows:

Command Format: **test** expression

expression - composed of constants, variables, and operators

Expressions will be looked at in greater detail later with some examples. There are a few items that need to be mentioned that apply to expressions. Expressions can contain one or more evaluation criteria that test will evaluate. A -a that separates two criteria is a logical AND operator. In this case, both criteria must evaluate to true in order for test to return a value of true. The -o is the logical OR operator. When this operator separates two criteria, one or the other (or both) must be true for test to return a true condition.

You can negate any criterion by preceding it with an exclamation mark (!). Parentheses can be used to group criteria. If there are no parentheses, the -a (logical AND operator) takes precedence over the -o (logical OR operator). The test utility will evaluate operators of equal precedence from left to right.

Within the expression itself, you must put special characters, such as parentheses, in quote marks so the BourneShell will not evaluate them but will pass them to test.

Since each element (evaluation criterion, string, or variable) in an expression is a separate argument, each must be separated by a space.

The test utility will work from the command line but it is more often used in a script to test input or verify access to a file.

Another way to do the test evaluation is to surround the expression with left and right brackets. A space character must appear after the left bracket and before the right bracket.

**test** expression = [ expression ]

**Test on Numeric Values**

Test expressions can be in many different forms. The expressions can appear as a set of evaluation criteria. The general form for testing numeric values is:

int1 op int2

This criterion is true if the integer int1 has the specified algebraic relationship to integer int2.

The valid operators (op) are:

-eq equal

-ne not equal

-gt greater than

-lt less than

-ge greater than or equal

-le less than or equal

**Test on Character Strings**

The evaluation criterion for character strings is similar to numeric comparisons. The general form is:

string1 op string2

The operators (op) are:

string1 = string2 true if string1 and string 2 are

equal

string1 != string2 true if string1 and string2 are not

equal

string1 true if string1 is not the null

string

**Sample Session:**

$ **cat** test\_string

number=1

numero=0001

if test $number = $numero

then echo "String vals for $number and $numero are ="

else echo "String vals for $number and $numero not ="

fi

if test $number -eq $numero

then echo "Numeric vals for $number and $numero are ="

else echo "Numeric vals for $number and $numero not ="

fi

$**chmod** 755 test\_string

$**sh -x** test\_string

number=1

numero=0001

+ test 1 = 0001

+ echo String vals for 1 and 0001 not =

String vals for 1 and 0001 not =

+ test 1 -eq 0001

+ echo Numeric vals for 1 and 0001 are =

Numeric vals for 1 and 0001 are =

$test\_string

String vals for 1 and 0001 not =

Numeric vals for 1 and 0001 are =

$

**Test on File Types**

The test utility can be used to determine information about file types. All of the criterion can be found in Appendix B. A few of them are listed here:

-r filename true if filename exists and is readable

-w filename true if filename exists and is writable

-x filename true if filename exists and is executable

-f filename true if filename exists and it is a plain file

-d filename true if filename exists and it is a directory.

-s filename true if filename exits and it contains information (has a size greater than 0 bytes)

**Example:**

$**test** -d new\_dir

If new\_dir is a directory, this criterion will evaluate to true.

If it does not exist, then it will be false.

**if then**

The format for this construct is:

Command Format:

**if** expression

**then** commands

**fi**

The if statement evaluates the expression and then returns control based on this status. The fi statement marks the end of the if, notice that fi is if spelled backward.

The if statement executes the statements immediately following it if the expression returns a true status. If the return status is false, control will transfer to the statement following the fi.

**Sample Session:**

$**cat** check\_args

if (test $# = 0)

then echo 'Please supply at least 1 argument'

exit

fi

echo 'Program is running'

$

This little script will check to insure that you are giving at least one argument. If none are given it will display the error message and exit. If one or more arguments are given it will display "Program is running" and run the rest of the script, if any.

**Sample Session:**

$check\_args

Please supply at least 1 argument

$check\_args xyz

Program is running

$

**if then else**

The format for this construct is:

Command Format:

**if** expression

**then** commands

**else** commands

**fi**

The else part of this structure makes the single-branch if statement into a two-way branch. If the expression returns a true status, the commands between the then and the else statement will be executed. After these have been executed, control will start again at the statement after the fi.

If the expression returns false, the commands following the else statement will be executed.

**Sample Session:**

$**cat** test\_string

number=1

numero=0001

if test $number = $numero

then echo "String values of $number and $numero are equal"

else echo "String values of $number and $numero not equal"

fi

if test $number -eq $numero

then echo "Numeric values of $number and $numero are equal"

else echo "Numeric values of $number and $numero not equal"

fi

Let's follow the execution of this script with tracing.

**Sample Session:**

$sh -x test\_string

number=1

numero=0001

+ test 1 = 0001

+ echo String values of 1 and 0001 are not equal

String values of 1 and 0001 are not equal

+ test 1 -eq 0001

+ echo Numeric values of 1 and 0001 are equal

Numeric values of 1 and 0001 are equal

$chmod a+x test\_string

$test\_string

String values of 1 and 0001 are not equal

Numeric values of 1 and 0001 are equal

$

**if then elif**

The format for this construct is:

Command Format:

**if** expression

**then** commands

**elif** expression

**then** commands

**else** commands

**fi**

The elif construct combines the else and if statements and allows you to construct a nested set of if then else structures.

**for**

The format for this construct is:

Command Format:

**for** loop-index in argument-list

**do**

commands

**done**

This structure will assign the value of the first item in the argument list to the loop index and executes the commands between the do and done statements. The do and done statements indicate the beginning and end of the for loop.

After the structure passes control to the done statement, it assigns the value of the second item in the argument list to the loop index and repeats the commands. The structure will repeat the commands between the do and done statements once for each argument in the argument list. When the argument list has been exhausted, control passes to the statement following the done.

**Sample Session:**

$**cat** find\_henry1

for x in project1 project2 project3

do

grep henry $x

done

**Sample Session:**

$head project?

==> project1 <==

henry

joe

mike

sue

==> project2 <==

joe

mike

sue

==> project3 <==

joe

mike

sue

henry

==> project4 <==

joe

mike

$find\_henry

henry

henry

$

Each file in the argument list was searched for the string, henry.

When a match was found, the string was printed.

**while**

The format for this construct is:

Command Format:

**while** expression

**do**

commands

**done**

As long as the expression returns a true exit status, the structure continues to execute the commands between the do and the done statement. Before each loop through the commands, the structure executes the expression. When the exit status of the expression is false (non-zero), control is passed to the statement following the done statement. The commands to be executed must change the expression test or an infinite loop can result.

**until**

The format for this construct is:

Command Format:

**until** expression

**do**

commands

**done**

The until and while structures are very similar. The only difference is that the test is at the top of the loop. The until structure will continue to loop until the expression returns true or a non-error condition. The while loop will continue as long as a true or non-error condition is returned.

The commands to be executed must change the expression test or an infinite loop can result.

**Sample Session:**

$**cat** until\_ex

secretname='jenny'

name='noname'

echo 'Try to guess the secret name!'

echo

until (test "$name" = "$secretname")

do

echo 'Your guess: \c'

read name

done

echo 'You did it!'

$

The until loop will continue until name is equal to the secret name.

**Sample Session:**

$chmod a+x until\_ex

$until\_ex

Try to guess the secret name!

Your guess: gaylan

Your guess: art

Your guess: richard

Your guess: jenny

You did it!

$

**case**

The format for this construct is:

Command Format:

**case** test-string in

pattern-1 ) commands-1 ;;

pattern-2 ) commands-2 ;;

pattern-3 ) commands-3 ;;

. .

. .

. .

\*) commands ;;

**esac**

The case structure allows a multiple-branch decision mechanism. The path that is taken depends on a match between the test-string and one of the patterns.

**Sample Session:**

$**cat** case\_ex

echo 'Enter A, B, or C: \c'

read letter

case $letter in

A) echo 'You entered A' ;;

B) echo 'You entered B' ;;

C) echo 'You entered C' ;;

\*) echo 'You did not enter A, B, or C' ;;

esac

$**chmod** a+x case\_ex

$case\_ex

Enter A, B, or C: B

You entered B

$case\_ex

Enter A, B, or C: b

You did not enter A, B, or C

$

This example uses the value of a character that the user entered as the test string. The value is represented by the variable letter. If letter has the value of A, the structure will execute the command following A. If letter has a value of B or C, then the appropriate commands will be executed. The asterisk indicates any string of characters; and it, therefore, functions as a catchall for a no-match condition. The lowercase b in the second sample session is an example of a no match condition.

444444444444444