# MODULE 1: UNIX ARCHITECTURE AND COMMAND USAGE GENERAL PURPOSE UTILITIES ESSENTIAL SYSTEM ADMINISTRATION

* 1. Introduction
  2. Unix Architecture
  3. Features of Unix
  4. POSIX and Single Unix Specification
  5. The Login Prompt
  6. Command Structure
     1. Options
     2. Filename Arguments
     3. Exceptions
  7. Understanding of Some Basic Commands
     1. echo
     2. printf
     3. ls
     4. who
     5. date
     6. passwd
     7. cal
  8. Flexibility of Command Usage
  9. Internal and External Commands
  10. type
  11. man
      1. man: Browsing the Manual Pages Online
      2. Understanding the man Documentation
      3. Using man to Understand man
      4. Further Help with man –k, apropos and whatis
  12. more
  13. Knowing the User Terminal, Displaying its Characteristics and Setting Characteristics
      1. uname : Knowing your Machines Characteristics
      2. tty: Knowing your Terminal
      3. stty: Displaying and Setting Terminal Characteristics
  14. Managing the Nonuniform Behaviour of Terminals and Keyboards
  15. Types of Accounts
      1. The root Login
      2. su: Becoming Super user
  16. /etc/passwd and /etc/shadow Files
  17. Managing sers and Groups
      1. groupadd
      2. Commands to Add, Modify and Delete users
         1. useradd
         2. usermod
         3. userdel

# MODULE 1: UNIX ARCHITECTURE AND COMMAND USAGE

### Introduction

* UNIX is an operating system.
* An operating system is a set of programs that act as a link between the computer and the user.
* The operating system (OS) manages the resources of a computer.
* Examples of computer resources are: CPU, RAM, disk memory, printers, displays, keyboard, etc.

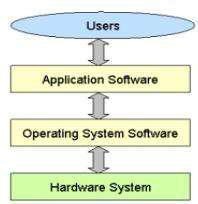


Figure 1.1 : Operating System

* UNIX OS allows complex tasks to be performed with a few keystrokes.
* UNIX OS doesn‘t tell or warn the user about the consequences of the command.
* Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
* There are various Unix variants available in the market.
* Solaris Unix, AIX, HP Unix and BSD are a few examples.
* Linux is also a flavor of Unix which is freely available.

### Unix Architecture (Components)

* The UNIX operating system (OS) consists of a kernel layer, a shell layer , an application layer & files.

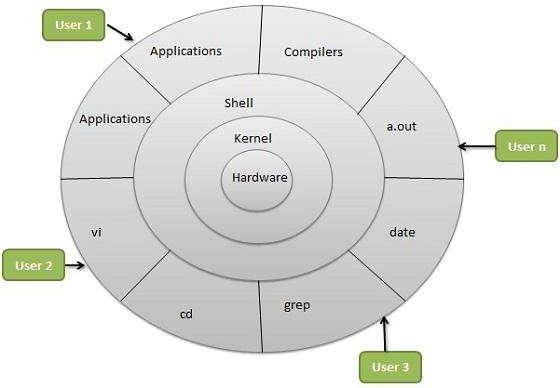


Figure 1.2 : Unix Architecture

### Kernel

* The kernel is the heart of the operating system.
* It interacts with the machine‘s hardware.
* It is a collection of routines written in C.
* It is loaded into memory when the system is booted.
* Main responsibilities:
  + 1. Memory management
    2. Process management (using commands: kill, ps, nohup)
    3. File management (using commands: rm, cat, ls, rmdir, mkdir)
* To access the hardware, user programs use the services of the kernel via system calls.

### System Call

* + A system call is a request for the operating system to do something on behalf of the user's program.
  + The system calls are functions used in the kernel itself.
  + To the programmer, the system call appears as a normal C function call.
  + UNIX system calls are used to manage the file system and control processes.  Example: read(), open(), close(), fork(), exec(), exit()
* There can be only one kernel running on the system.
* The shell interacts with the user.
* The shell is a command line interpreter (CLI).
* Main responsibilities:

1. interprets the commands the user types in and
2. dispatches the command to the kernel for execution

* There can be several shells in action, one for each user who‘s logged in.
* There are multiple shells that are used by the UNIX OS.
* For example: Bourne shell (sh), the C shell (csh), the Korn shell (ksh) and Bourne Again shell (bash).

### Application

* This layer includes the commands, word processors, graphic programs and database management programs.

### File

* A file is an array of bytes that stores information.
* All the data of Unix is organized into files.
* All files are then organized into directories.
* These directories are further organized into a tree-like structure called the filesystem.

### Features of Unix

**1) Multiuser**

* UNIX is a multiprogramming system.
* Multiple users can access the system by connecting to points known as terminals.
* Several users can run multiple programs simultaneously on one system.
* UNIX is a multitasking system.
* A single user can also run multiple tasks concurrently.
* For example:

→ edit a file

→ print another file one on the printer

→ send email to a friend and

→ browse www

* The kernel is designed to handle a user's multiple needs.
* In a multitasking environment, a user sees one job running in the foreground; the rest run in the background.
* User can switch jobs between background and foreground, suspend, or even terminate them.
* UNIX has very sophisticated pattern matching features.
* Regular Expressions are a feature of UNIX.
* Regular Expressions describe a pattern to match, a sequence of characters, not words, within a line of text.

### Portable

* UNIX can be installed on many hardware platforms.
* Unix operating system is written in C language, hence it is more portable than other operating systems.

### UNIX Toolkit

* UNIX offers facility to add and remove many applications as and when required.
* Tools include

→ general purpose tools

→ text manipulation tools

→ compilers/interpreters

→ networked applications and

→ system administration tools

* The UNIX shell is also a programming language; it was designed for programmer, not for end user.
* It has all the necessary ingredients, like control structures, loops and variables, that establish powerful programming language.
* This features are used to design shell scripts – programs that can also invoke UNIX commands.
* Many of the system's functions can be controlled and automated by using these shell scripts.
* The principal on-line help facility available is the man command, which remains the most important references for commands and their configuration files.
* Apart from the man documentation, there's a vast ocean of UNIX resources available on the Internet.

### POSIX and Single Unix Specification

* The Portable Operating System Interface (POSIX) is a family of standards specified by IEEE for maintaining compatibility between operating systems.
* Two of the most important standards from POSIX are:
  + 1. POSIX.1 – Specifies the C application program interface – the system calls (Kernel)
    2. POSIX.2 – Deals with the Shell and utilities
* In 2001, a joint initiative of X/Open and IEEE resulted in the unification of two standards.
* This is the Single UNIX Specification, Version 3 (SUSV3).
* The ―Write once, adopt everywhere‖ approach to this development means that once software has been developed on any POSIX machine it can be easily ported to another POSIX compliant machine with minimum or no modification.

### The Login Prompt How to log in

* Have your userid (user identification) and password ready.
* Type your userid at the login prompt, then press ENTER. Your userid is case sensitive, so be sure you type it exactly as your system administrator has instructed.
* Type your password at the password prompt, then press ENTER. Your password is also case sensitive.
* If you provide the correct userid and password, then you will be allowed to enter into the system.
* While the prompt($) is displayed, you can type a command (say date). login: kumar

kumar's password: \*\*\*\*\*

Last login: Sun Jun 14 09:32:32 2017 from 162.61.164.73

$ date

Thu Jun 25 08:30:19 MST 2017

### Command Structure

* UNIX commands take the following general

< form: verb [options] [arguments] >

where verb is the command name that can take a set of optional options and one or more optional arguments.

* Commands, options and arguments have to be separated by spaces or tabs to enable the shell to interpret them as words.
* A contiguous string of spaces and tabs together is called a whitespace.
* The shell compresses multiple occurrences of whitespace into a single whitespace.

### Options

* An option is preceded by a minus sign (-) to distinguish it from filenames.
* Example:

$ ls –l // -l option list all the attributes of the file note

* There must not be any whitespaces between – and l.
* Options are also arguments, but given a special name because they are predetermined.
* Options can be normally combined with only one – sign.

thus $ ls –l –a –t is same as $ ls –lat

* Because UNIX was developed by people who had their own ideas as to what options should look like, there will be variations in the options.
* Some commands use + as an option prefix instead of -.

### Filename Arguments

* Many UNIX commands use a filename as argument so that the command can take input from the file.
* If a command uses a filename as argument, it will usually be the last argument, after all options.
* Example:

ls -lat chap01 chap02 chap03 # Multiple filenames as arguments cp file1 file2 file3 dest dir

rm file1 file2 file3

* The command with its arguments and options is known as the command line.
* This line can be considered complete only after the user has hit [Enter].
* The complete line is then fed to the shell as its input for interpretation and execution.

### Exceptions

* There are some commands that don't accept any arguments.
* There are also some commands that may or may not be specified with arguments.
* For Example:

→ without arguments (ls)

→ with only options (ls –l)

→ with only filenames (ls f1 f2), or

→ using a combination of both (ls –l f1 f2).

* There are some commands compulsorily take options (cut).
* There are some commands which can take an expression as an argument, or a set of instructions as argument. Ex: grep, sed

### Understanding of Some Basic Commands

* + 1. **echo**
* This command can be used to display a message on the terminal.
* Example:

$ echo "Welcome to UNIX \n" Welcome to UNIX

* This command can be used with following escape characters:

|  |  |
| --- | --- |
| "\a" | Audible Alert (Bell) |
| "\b‟ | Back Space |
| "\f‟ | Form Feed |
| "\n‟ | New Line |
| "\r‟ | Carriage Return |
| "\t‟ | Horizontal Tab |
| "\v‟ | Vertical Tab |
| "\\‟ | Backslash |

### printf

* This command can be used to display a message on the terminal.
* This command can be used with following escape characters:

|  |  |
| --- | --- |
| "\a" | Audible Alert (Bell) |
| "\b‟ | Back Space |
| "\f‟ | Form Feed |
| "\n‟ | New Line |
| "\r‟ | Carriage Return |
| "\t‟ | Horizontal Tab |
| "\v‟ | Vertical Tab |
| "\\‟ | Backslash |

* Example:

$ printf "Welcome to UNIX \n" Welcome to UNIX

* Similar to C language, this command can be used with following format specifiers:

|  |  |
| --- | --- |
| %d | Decimal integer |
| %f | Floating point number |
| %s | String |
| %o | Octal integer (base 8) |
| %x | Hexadecimal integer (base 16) |

* Syntax:

printf("format-string", variable-list);

where format-string contains one or more format-specifiers variable-list contains names of variables

* Example:

$ printf "My current shell is %s\n" $SHELL My current shell is /bin/ksh

### ls

* ls command can be used to obtain a list of all filenames in the current directory.
* -l option can be used to obtain a detailed list of attributes of all files in the current directory.
* Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| $ ls -l  Type & Perm Link Owner | Group | Size | Date & Time | File Name |
| -rwxr-xr-- 1 kumar | metal | 195 | may 10 13:45 | chap01 |
| drwxr-xr-x 2 kumar | metal | 512 | may 09 12:55 | helpdir |

### who

* This command can be used to display the details of all the users logged-in into the unix system at

|  |  |  |
| --- | --- | --- |
| the same time.   * Example: |  | |
| $ who  kumar | tty0 | Oct 8 14:10 |
| rama | tty2 | Oct 4 09:08 |
| * –H option can be used to display the header information. * Example: | | |
| $ who |  |  |
| NAME | LINE | TIME COMMENT |
| kumar | tty0 | Oct 8 14:10 |
| rama | tty2 | Oct 4 09:08 |

* -u option can be used to display detailed information of users.
* Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| $ who -Hu  NAME | LINE | TIME | IDLE | PIDCOMMENT |
| kumar | tty0 | Oct 8 14:10 | 00:18 | 185 |
| rama | tty2 | Oct 4 09:08 | 00:23 | 123 |

* + 1. **date**
* This command can be used to display the current date and time.
* Example:

$ date

Mon Sep 4 16:40:02 IST 2017

* This command can also be used with suitable format specifiers as arguments.
* Following are some format specifiers:

d – day of month (1 - 31) m - Month (01-12) y – last two digits of the year. H– hour (00-24) M – minute (00-59) S – second (00-59)

$ date +%format specifier

* Example:

$ date +"%d-%m-%y" 04-09-17

$ date +%m // displays month number 09

### passwd

* This command can be used to change user password.
* All Unix systems require passwords to help ensure that your files and data remain secure from hackers.
* Following are the steps to change your password –
  + - 1. To start, type password at the command prompt as shown below.
      2. Enter your old password, the one you're currently using.
      3. ype in your new password.
      4. You must verify the password by typing it again.

$ passwd

Changing password for kumar (current) Unix password: \*\*\*\*\* New Unix password: \*\*\*\*\* Retype new Unix password: \*\*\*\*\*

passwd: all authentication tokens updated successfully

$

### cal

* This command can be used to display the calendar of the current month.
* Syntax:

cal [ [ month] year ]

* Example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| $ cal  September 2017 | | | | | | |
| Su | Mo | Tu | We | Th | Fr | Sa |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 |  |  |  |  |  |

* This command can also be used to display the calendar of any specific month or a complete year.
* Example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| $ cal 9 2017  September 2017 | | | | | | |
| Su | Mo | Tu | We | Th | Fr | Sa |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 |  |  |  |  |  |

* You can't hold the calendar of a year in a single screen page; it scrolls off rapidly to end of the year.
* To pause at each screen page, a pipe "|" can be connected to more pager:
* Example:

$cal 2017 | more

### Flexibility of Command Usage

* UNIX provides flexibility in using the commands.
* A command can often be entered in more than one way.
* Shell allow the following type of command usage:
  + 1. Combining Commands.
    2. A command line can overflow or Be split into multiple lines.
    3. Entering a command before previous command has finished.

### Combining Commands

* UNIX allows you to specify more than one command in the single command line.
* Example:

$ wc sample.txt ; ls –l sample.txt //Two commands separated by ; (semicolon). 2 3 16 sample.txt

-rw-rw-r-- 1 kumar group 16 Jan 30 09:35 sample.txt

$ ls | wc //Two commands combined here using filter

* You can even group several commands together so that their combined output is redirected to a file. (wc sample.txt ; ls –l sample.txt) > newfile
* When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. Here ; is known as a metacharacter.

### A Command line can be split into multiple lines

* UNIX terminal width is restricted to maximum 80 characters.
* Shell allows command line to overflow or be split into multiple lines.
* Example:

$ echo ―This is // $ first prompt

* a three-line // > is a secondary prompt
* text message‖ // Command line ends here This is a three-line text message

### Entering a Command before previous command has finished

* You need not have to wait for the previous command to finish before you can enter the next command.
* Subsequent commands can be entered at the keyboard without waiting for prompt.
* The input remains stored in a buffer maintained by kernel for all keyboard input.
* The next command is passed on to the shell for interpretation after the previous program has completed.

### Internal and External Commands

* Some commands are implemented as part of the shell itself rather than separate executable files. Such commands that are built-in are called internal commands.
* If the command (file) has an independence existence in the /bin directory, it is called external command.
* Examples:

$ type echo // echo is an internal command echo is shell built-in

$ type ls // ls is an external command ls is /bin/ls

* If the command exists both as an internal and external one, shell execute internal command only.
* Internal commands will have top priority compare to external command of same name.

### type

* The UNIX is command based system i.e.,- things happens because the user enters commands in.
* Usually, UNIX commands are less than 5 characters long.
* All UNIX commands are single words like – cat, ls, pwd, date, mkdir, rmdir, cd, grep etc.
* The command names are all in lowercase.
* Example:

$ LS bash:

LS: command not found

* All UNIX commands are files containing programs, mainly written in C.
* All UNIX commands(files) are stored in directories(folders).
* If you want to know the location of executable program (or command), use type command.
* Example:

$ type ls

ls is /bin/ls

* When you execute ls command, the shell locates this file in the /bin directory and makes arrangements to execute it.

### man

* + 1. **man: Browsing The Manual Pages Online**
* This command can be used to display manual (documentation) of a specified command.
* Syntax:

man command

* A pager is a program that

→ displays one screenful information and

→ pauses for the user to view the contents.

* The man command is configured to work with a pager.
* Following two commands can be used for navigation:
  + - 1. Spacebar or f – moves forward one screen
      2. b – moves back one screen
* Finally, to quit the pager, press q. You'll be returned to the shell's prompt ($).

### Understanding The man Documentation

* The man documentation is organized in eight (08) sections.
* Example:

$ man wc //Help on the wc command User Commands wc(1)

### NAME

wc – display a count of lines, words and characters in a file **SYNOPSIS**

wc [ -c | -m | -C] [ -lw ] [ files. ]

### DESCRIPTION

The wc utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.

### OPTIONS

The following options are supported:

-c Count Bytes

-m Count Characters

-l Count Lines

-w Count Words

### OPERANDS

The following operands are supported:

File - a path name of an input file. If no file operand are specified, the standard input will be used.

### USAGE

See largefiles(5) for the behavious of wc when encountering files >= 2 Gbytes.

### EXIT STATUS

0 Successful Completion

* 0 An Error Occurred

### SEE ALSO

space(3C), iswalpha(3C), iswspace(3C), setlocale(3C), attributes(5), environ(5), largefile(5)

* A man is divided into a number of compulsory and optional sections.
* Every command doesn‘t need all sections, but the first three (NAME, SYNOPSIS and DESCRIPTION) are generally seen in all man pages.

1. NAME presents a one-line introduction of the command.
2. SYNOPSIS shows the syntax used by the command.
3. DESCRIP ION provides a detailed description.

* The SYNOPSIS follows certain conventions and rules:

1. If a command argument is enclosed in rectangular brackets, then it is optional; otherwise, the argument is required.
2. The ellipsis (a set if three dots) implies that there can be more instances of the preceding word.
3. The | means that only one of the options shows on either side of the pipe can be used.

* All the options used by the command are listed in OPTIONS section.
* There is a separate section named EXIT STATUS which lists possible error conditions and their numeric representation.

### Using man to Understand man

* You can use man command to view its own documentation.
* Example:

$ man man //Viewing man pages with man

* You can also set the pager to use with man ($ PAGER=less ; export PAGER).
* To understand which pager is being used by man, use $ echo $PAGER.

### Further Help with man –k, apropos and whatis

* man –k: Searches a summary database and prints one-line description of the command.
* Example:

$ man –k awk //To know what awk does

awk (1) - pattern scanning and text processing language m awk (1) - pattern scanning and text processing language n awk (1) - pattern scanning and text processing language

* apropos can be used to list the commands and files associated with a keyword.
* Example:

awk (1) - pattern scanning and text processing language m awk (1) - pattern scanning and text processing language n awk (1) - pattern scanning and text processing language

//Same as $ man -k awk

$ apropos awk

* Example:

$ whatis awk // Lists one-line description of command and same as $man -f awk awk

(1) - pattern scanning and text processing language

### more

* more command can be used to display the content of a file on the screen, one page at a time.
* After each page, it stops and waits for you to tell it what to do.
* If the file contents is more, it will show the filename and percentage of the file that has been viewed:

----More--- (15%)

* Syntax:

more FILENAME

### Navigation

* Following two commands can be used for navigation:
  + 1. Spacebar or f – moves forward one screen
    2. b – moves back one screen
* Finally, to quit the pager, press q.
* You'll be returned to the shell's prompt($).

### Repeat Factor

* The repeat factor used as a command prefix to repeat the command as many times as the prefix.
* For example:

→ 10f moves forward 10 screen. Here, 10 is acting as a repeat factor.

### Searching for a pattern

* A pattern can be searched in a file using following 2 commands.

|  |  |
| --- | --- |
| **Command** | **Function** |
| /Pattern | searches forward for pattern in the file. |

* Syntax:

/Pattern [Enter]

### Repeating the Last Pattern Search

* The previous search command can be repeated using following 2 commands:

|  |  |
| --- | --- |
|  |  |
| n | repeats the previous search command in the same direction |

* n repeats search in same direction of original search.

### Using more in pipeline

* Example:

ls | more

* Here, pipe (|) is used where the output of one command is used as the input of the other command.

Table: Internal commands of more and less

|  |  |  |
| --- | --- | --- |
| **more** | **less** | **action** |
| spacebar or f | spacebar or f | one page forward |
| 10f | - | 20 pages forward |
| b | b | one page backward |
| 10b | - | 20 pages backward |
| [Enter] | [Enter] | one line forward |
| - | k | one line back |
| - | p or 1G | beginning of file |
| - | G | end of file |
| /Pattern | /Pattern | searches forward for pattern |
| n | n | repeat search forward |
| - | ?Pattern | searches backward for pattern |
| .(dot) | - | repeat last command |
| v | v | start vi editor |
| !cmd | !cmd | executes unix command |
| q | q | quit |
| h | h | help |

# MODULE 1(CONT.): GENERAL PURPOSE UTILITIES

### Knowing the User Terminal, Displaying its Characteristics and Setting Characteristics

**1.13.1 uname : Knowing your Machines Characteristics**

* This command can be used to display certain features of the OS running on your machine.
* Example:

$ uname -s

SunOS //name of OS used by sun solaris

$ uname -R

5.8 //version of OS

### tty: knowing your terminal

* This command can be used to know device name of the terminal.
* Example:

$ tty

/dev/pts/xyz //here treminal filename is xyz resident in pts directory

//which is in turn in /dev directory

### stty: Displaying and Setting Terminal Characteristics

* This command can be used to display and set various terminal attributes.
* –a option can be used to display the current settings.

$ stty -a

speed 38400 baud; rows 24; columns 116; intr = ^C; quit = ^\; erase = ^?; kill = ^U;

### Changing the Settings

* To remove the backspacing, we can use the following command:

$stty –echoe

* To remove echo command to work, we can use the following command:

$stty –echo

### Changing the Interrupt Key (intr)

* To change the interrupt setting, we can use the following command:

$stty intr \^c

### Changing the end-of-File Key (eof)

* To change the end-of-File key setting, we can use the following command: $stty eof \^a
* [ctrl-a] will now terminate input for those commands that expects input from the keyboard.

### When everything Else fail (sane)

* stty also provides another argument to set the terminal characteristics to value that will work on most terminals.
* Use the word sane as a single argument to the command:

$stty sane //restores sanity to the terminal

### Managing the Non-uniform Behaviour of Terminals and Keyboards

* Terminals and keyboards have no uniform behavioral pattern.
* Terminal settings directly impact the keyboard operation.
* The following table lists keyboard commands to try when things go wrong.

|  |  |
| --- | --- |
| **Keystroke or Command** | **Function** |
| [Ctrl-h] | Erases text |
| [Ctrl-c] or Delete | Interrupts a command |
| [Ctrl-d] | Terminates login session or a program that expects its input from  keyboard |
| [Ctrl-s] | Stops scrolling of screen output and locks keyboard |
| [Ctrl-q] | Resumes scrolling of screen output and unlocks keyboard |
| [Ctrl-u] | Kills command line without executing it |
| [Ctrl-\] | Kills running program but creates a core file containing the memory  image of the program |
| [Ctrl-z] | Suspends process and returns shell prompt; use fg to resume job |
| [Ctrl-j] | Alternative to [Enter] |
| [Ctrl-m] | Alternative to [Enter] |
| stty sane | Restores terminal to normal status |

# MODULE 1(CONT.): ESSENTIAL SYSTEM ADMINISTRATION

### Types of Accounts

There are 2 types of accounts on a Unix system:

### 1) Root Account

* The system administrator is known as superuser or root user.
* The superuser has complete control of the system.
* User accounts provide interactive access to the system for users and groups of users.
* General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

### The root Login

* The system administrator is known as superuser or root user.
* The superuser has complete control of the system.
* The superuser can run any commands without any restriction.
* The job of superuser includes:

→ maintaining user accounts

→ providing security and

→ managing disk space

→ performing backups

* The root account doesn‘t need to be separately created but comes with every system.
* The root account's password is generally set at the time of installation of the system and has to be used on logging in:

Login: root

Password: \*\*\*\*\*\*\*\*\* [Enter] # -

* The command prompt of root is hash (#).
* Once you login, you are placed in root‘s home directory "/".
* /sbin and /usr/sbin contains administrative commands of the system.

### su: Becoming Super User

* Any user can acquire superuser status with the su command if they know the root password.
* For example, the user "kumar" becomes a superuser in this way:

$ su

Password: \*\*\*\*\* # pwd

/home/kumar

* Though the current directory doesn‘t change, the # prompt indicates that kumar now has powers of a superuser.
* To be in root‘s home directory on superuser login, use su –l

### Creating a User’s Environment

* User‘s often rush to the administrator with the complaint that a program has stopped running.
* The administrator first tries running it in a simulated environment.
* su command with a – (hyphen) can be used to recreate the user‘s environment without the login- password.

su –kumar

* This sequence executes kumar‘s .profile and temporarily creates kumar‘s environment.
* su runs in a separate sub-shell, so this mode is terminated by hitting [ctrl-d] or using exit.

### /etc/passwd and /etc/shadow Files

* There are four main user administration files:
  + 1. /etc/passwd: Keeps the user account and password information. This file holds the majority of information about accounts on the Unix system.
    2. /etc/shadow: Holds the encrypted password of the corresponding account.
    3. /etc/group: This file contains the group information for each account.
    4. /etc/gshadow: This file contains secure group account information.

### /etc/passwd

* This file contains following user account information:
  + This is the name used for logging into a UNIX system.

### Password

* This stores the encrypted password which looks like **\*\*\*\*\*\*.**

### UID

* This is user‘s numerical identification.
* No two users can have the same UID.

### GID

* This is user's numerical group identification.

### Comments or GCOS

* This is contains user details or name address.
* This name is used at the front of the email address for this user.

### Home Directory

* The directory where the user ends up on logging in.
* The login program reads this field to set the variable HOME.

### Login shell

* This is the first program executed after logging in.
* This is usually the shell (/bin/ksh).
* The login program reads this field to set the variable SHELL.

### 2) /etc/shadow

* This file contains encrypted password of the corresponding account.
* This is the control file used by passwd to verify the correctness of a user's password.
* For every line in /etc/passwd, there‘s a corresponding entry in /etc/shadow.
* As a regular user, you do not have **read** or **write** access to this file for security reasons.
* Only superuser can access this file.

### Managing Users and Groups

* + 1. **groupadd**
* This command can be used to create a new group.
* We need to create groups before creating any account.
* Syntax:

groupadd -g gid groupname

where -g GID → The numerical value of the group's ID groupname → Actual group name to be created

* Example:

groupadd -g 192 ISE //192 is the GID for ISE group

* /etc/group: This file contains the group information for each account.
* Group usually has more than one member with a different set of privileges.
* Creating a group involves defining the following parameters:
  + - 1. A User identification number (UID) and username
      2. A group identification number (GID) and groupname
      3. The home directory 4) The login shell 5) The mailbox in /var/mail
* /etc/passwd: This file contains above 5 user account information.

### Commands to Add, Modify and Delete Users

* Following commands can be used to create and manage user accounts:
  + - 1. useradd 2) usermod & 3) userdel

### useradd

* useradd command can be used to create a new user.
* Syntax:

useradd -u userid -g groupname -d homedir -s shell -m accountname where -u userid → You can specify a user id for this account

-g groupname → Specifies a group account for this account -d homedir → Specifies home directory for the account

-s shell → Specifies the default shell for this account

-m → Creates the home directory if it doesn't exist accountname → Actual account name to be created

* Example:

# useradd -u 999 -g ISE -d /home/USP -s /bin/ksh -m kumar

* This creates the user "kumar" with UID(userid)=999 group name= ISE

home directory = /home/kumar shell = Korn shell MAIL = /var/mail

* This command

→ modifies the /etc/passwd, /etc/shadow, and /etc/group files and

→ creates a home directory.

* The .profile file is created and copied to user‘s home directory.
* Then, passwd command can be used to set new user password.

### usermod

* The usermod command can be used to make changes to an existing account.
* This command uses the same options as the useradd command.

# usermod -s /bin/bash kumar // to set current shell to bash shell # usermod -g CSE kumar // to set current group to CSE

### userdel

* The userdel command can be used to delete an existing user & his account. # userdel kumar // to delete user "kumar"
* Example: To illustrate the usage of useradd, usermod & userdel.

# useradd -u 999 -g ISE -d /home/USP -s /bin/ksh -m kumar

# usermod -s /bin/bash kumar // to set current shell to bash shell # usermod -g CSE kumar // to set current group to CSE

# userdel kumar // to delete user "kumar"

# MODULE 2: THE FILE SYSTEM

**HANDLING ORDINARY FILES BASIC FILE ATTRIBUTES**

* 1. UNIX Files
  2. Naming Files
  3. Basic File-Types
     1. Ordinary (Regular) File
     2. Directory File
     3. Device File
  4. Parent Child Relationship
  5. Standard Directories
  6. HOME Variable
  7. PATH Variable
  8. Relative and Absolute Pathname
     1. Absolute Pathname
     2. Relative Pathname
  9. Directory Commands
     1. pwd
     2. cd
     3. mkdir
     4. rmdir
  10. File Related Commands
      1. cat
      2. wc
      3. cp
      4. mv
      5. rm
      6. od
  11. File Attributes and Permissions and Knowing them
      1. Listing Directory Attributes
  12. ls
      1. ls Options
  13. Changing File Permissions
      1. Relative Permissions
      2. Absolute Permissions
      3. Changing File Ownership
         1. chown
         2. chgrp
      4. Directory Permissions

# MODULE 2: THE FILE SYSTEM

### UNIX Files

* A file is the container for storing information.
* The file doesn't store 1) file-size and 2) file-name.
* Some attributes of file are file-type, permissions, links, owner, group-owner etc.
* These file attributes are stored in inode table which is accessible only to kernel.
* A UNIX system makes no difference between a file and a directory, since a directory is just a file containing names of other files.
* Programs, services, texts, images etc are considered to be files.
* Generally, all devices including I/O devices are also considered to be files.

### Naming Files

* A filename can consist up to 255 characters.
* Files may or may not have extensions.
* Files consist of any ASCII character expect the "/" and NULL character.
* Following is recommended for filenames:

→ Alphabetic characters and numerals

→ period(.), hyphen(-) and underscore( )

* However, users are permitted to use control characters or other unprintable characters in a filename.
* UNIX is case sensitive; chap01, Chap01 and CHAP01 are three different filenames.

### Basic File Types

* Three categories of files are:
  + 1. Ordinary file (or regular file) – It contains only data as a stream of characters.
    2. Directory file – it contains files and other sub-directories.
    3. Device file – all devices and peripherals are represented by files.

### Ordinary (Regular) File

* An ordinary file is a file on the system that contains data, text, or program instructions.
* An ordinary file can be either a text file or a binary file.
  + - 1. A text file contains only printable characters and you can view and edit them.  Examples: All C and Java program sources, shell scripts are text files.
  + Every line of a text file is terminated with the newline character.
    - 1. A binary file contains both printable and non printable characters that cover the entire ASCII range.
  + Examples: Most Unix commands, executable files, pictures, sound and video files are binary files.

### Hidden Files

* + An invisible file is one, the first character of which is the dot or the period character (.).
  + Unix programs (including the shell) use most of these files to store configuration information.  Some common examples of the hidden files include the files:

.profile − The Bourne shell ( sh) initialization script.

.kshrc − The Korn shell ( ksh) initialization script.

.cshrc − The C shell ( csh) initialization script.

.rhosts − The remote shell configuration file.

### Directory File

* A directory contains no data, but keeps details of the files and subdirectories that it contains.
* A directory file contains one entry for every file and subdirectory that it houses.
* Each entry has two components namely,
  + - 1. filename and
      2. unique identification number of the file or directory (called the inode number).
* When you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (filename and inode number) associated with the file.
* Unix directories are equivalent to windows folders.

### Device File

* All the operations on the devices are performed by reading or writing the file representing the device.
* Advantage of device file is that some of the commands used to access an ordinary file also work with device file.
* Device filenames are generally found in a single directory structure, /dev.
* A device file is not really a stream of characters.
* It is the attributes of the file that entirely govern the operation of the device.
* The kernel identifies a de ice from its attributes and uses them to operate the device.

### Parent Child Relationship

* All data in Unix is organized into files.
* All files are organized into directories.
* These directories are organized into a tree-like structure called the filesystem.

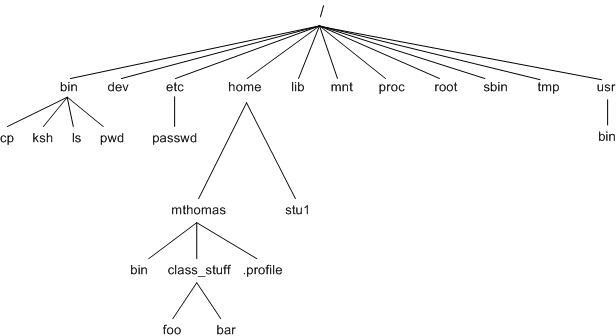


Figure 2.4: Parent Child Relationship

* The feature of UNIX file system is that there is a top, which serves as the reference point for all files.
* This top is called root (represented by a "/").
* The root is actually a directory.
* The root directory (/) has a number of subdirectories under it.
* The subdirectories in turn have more subdirectories and other files under them.
* Every file apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.
* In parent-child relationship, the parent is always a directory.

### Standard Directories (UNIX Filesystem)

* Following are the directories that exist on the major versions of Unix

1) / (root directory)

* This contains only the directories needed at the top level of the file structure. 2) /bin
* This contains common programs, shared by the system, the system administrator and the users.

1. /dev

* This contains references to all the CPU peripheral hardware, which are represented as files with special properties. These are device drivers.

1. /etc

* This contains system configuration files.

1. /home

* This contains home directories of the common users. 6) /lib
* This contains library files, including files for all kinds of programs needed by system and users. 7) /sbin
* This contains programs for use by the system and the system administrator.

1. /tmp

* This contains temporary space for use by the system. This space is cleaned upon reboot. So, don't use this for saving any work.

1. /usr

* This contains programs, libraries, documentation etc. for all user-related programs.

1. /var

* This contains storage for all variable files and temporary files created by users, such as

→ log files

→ mail queue or

→ print spooler area

### HOME Variable

* When a user logs into the system, the user will be placed in a directory called home directory.
* Home directory is created by the system when the user account is created.
* The shell-variable HOME indicates the home directory of the current user.
* This variable is set for a user by the system admin in /etc/passwd.
* Example:

$ echo $HOME

/home/kumar

* Here, absolute pathname is shown.
* Absolute pathname is a sequence of directory names starting from root (/).
* The subsequent slashes are used to separate the directories.

### PATH Variable

* This variable specifies the locations in which the shell should look for commands.
* Example:

$ echo $PATH

/bin: /usr/bin:

* When you specify a command like date, the system will locate the associated file from a list of directories specified in the PATH variable and then executes it.
* The PATH variable also includes the current directory.
* Whenever you enter any UNIX command, you are actually specifying the name of an executable file located somewhere on the system.
* The system goes through the following steps in order to determine which program to execute:
  + 1. Built in commands (such as cd and history) are executed within the shell.
    2. If an absolute path name (such as /bin/ls) or a relative path name (such as ./myprog), the system executes the program from the specified directory.
    3. Otherwise, the PATH variable is used.

### Relative and Absolute Pathname

* A pathname is a text string made up of one or more names separated by a "/".
* A pathname specifies how to traverse (navigate) the hierarchical directory names in the file system to reach some destination object.

### Absolute Pathname

* An absolute pathname begins with a slash (/).
* The Absolute path defines the location of a Directory or a file from the root file system (/).
* The absolute path contains the full path to the directory or file.
* The relative pathname do not begin with "/".
* It specify the location relative to your current working directory.
  + - 1. . (a single dot) - This represents the current directory.
      2. .. (two dots) - This represents the parent directory.
* Example:
* 'date' command can executed in two ways as follows:

|  |  |
| --- | --- |
| **Using Absolute Pathname** | **Using Relative Pathname** |
| $ /bin/date  Thu Sep 7 10:20:29 IST 2017 | $ date  Thu Sep 7 10:20:29 IST 2017 |

* Example:
* P1.java can be copied to kumar under home directory in two ways as follows:

|  |  |
| --- | --- |
| **Using Absolute Pathname** | **Using Relative Pathname** |
| $ pwd  /home/kumar  $ cp /home/sharma/P1.java /home/kumar | $ pwd  /home/kumar  $ cp /home/sharma/progs . |

* Example:
* cd & mkdir can be used in two ways as follows:

|  |  |
| --- | --- |
| **Using Absolute Pathname** | **Using Relative Pathname** |
| $ pwd | $ pwd |
| /home/kumar | /home/kumar |
| $ mkdir /home/kumar/progs | $ mkdir progs |
| $ cd /home/kumar/progs | $ cd progs |
| $ pwd | $ pwd |
| /home/kumar/progs | /home/kumar/progs |
| $ cd /home/kumar | $ cd .. |
| $ pwd | $ pwd |
| /home/kumar | /home/kumar |

### Directory Commands

* + 1. **pwd (print working directory)**
* This command can be used to display the current working directory.
* Example:

$ pwd

/home/kumar

* Like HOME, it displays the absolute path.

### cd

* This command can be used to change the current working directory.
* Syntax

cd PATHNAME

* This command can be used with the argument.
* This command can work with both absolute and relative path names.
* Example:

|  |  |
| --- | --- |
| **Using relative pathname** | **Using absolute pathname** |
| $ pwd | $ pwd |
| /home/kumar | /home/kumar |
| $ mkdir progs | $ mkdir /home/kumar/progs |
| $ cd progs | $ cd /home/kumar/progs |
| $ pwd | $ pwd |
| /home/kumar/progs | /home/kumar/progs |

### Case 2:

* This command can also be used without the argument.
* When used without argument, this command changes the working directory to home directory.
* Example:

$ pwd

/home/kumar/progs

$ cd

$ pwd

/home/kumar

### Case 3:

* This command can also be used with short hand notations.
* Example:

$ cd / // changes the working directory to root directory (/)

$ cd .. // changes the working directory to the one level up parent directory (..)

$ cd ../.. // changes the working directory to the two level up parent directory (../..)

### mkdir

* This command can be used to create a new directory.
* Syntax:

mkdir DIRECTORY\_NAME

* This command can work with both absolute and relative path names.
* Example:

|  |  |
| --- | --- |
| **Using relative pathname** | **Using absolute pathname** |
| $ pwd | $ pwd |
| /home/kumar | /home/kumar |
| $ mkdir progs | $ mkdir /home/kumar/progs |
| $ cd progs | $ cd /home/kumar/progs |
| $ pwd | $ pwd |
| /home/kumar/progs | /home/kumar/progs |

### Case 2:

* This command can also accept more than one directory name as arguments.
* Example:

$mkdir usp ade dms // creates 3 directories under current directory

$mkdir sem3 sem3/usp sem3/ade // creates 3 subdirectories – sem3, usp, ade

* The order of specifying arguments is important. You cannot create subdirectories before creation of parent directory.
* System refuses to create a directory due to following reasons:
  + - 1. User doesn‗t have permission to create directory. (i.e. directory write protected).
      2. The directory already exists.
      3. There may be ordinary file by that name in the current directory.

### rmdir

* This command can be used to delete a directory.
* Syntax:

rmdir DIRECTORY NAME

### Case 1:

* Example:

$ rmdir usp // delete usp directory under current directory

### Case 2:

* This command can also accept more than one directory name as arguments.
* Example:

$ mkdir sem3 sem3/usp sem3/ade // creates 3 subdirectories – sem3, usp, ade

$ rmdir sem3/ade sem3/usp sem3 // deletes 3 subdirectories – sem3, usp, ade

* Here, rmdir expects the arguments to be reverse of mkdir's arguments.
* The order of specifying arguments is important. You cannot delete parent directories before deletion of subdirectories.
* System refuses to delete a directory due to following reasons:
  + - 1. User doesn‗t have permission to delete directory. (i.e. write protected directory).
      2. The directory doesn‗t exist in system.
      3. The directory is your present working directory.

# MODULE 2(CONT.): HANDLING ORDINARY FILES

### File Related Commands

* + 1. **cat**
* This command can be used to display the content of a file on the terminal.
* Syntax:

cat FILENAME

// contents of P2.c

// contents of P1.c

$ cat P1.c WELCOME TO UNIX

$ cat P2.c WELCOME TO PERL

### Case 2:

* This command can also accept more than one filename as arguments.
* Example:

$ cat P1.c P2.c

WELCOME TO UNIX // contents of P1.c WELCOME TO PERL // contents of P2.c

* Here, the content of the second file is shown immediately after the first file.
* So, this command concatenates two files- hence its name (cat).
* This command can also be used to create a new file.
* Syntax:

cat > FILENAME

* Example:

// contents of P3.c

// contents of P3.c

// erminates P3.c

$ cat > P3.c WELCOME TO SHELL

[ctrl-d]

$ cat P3.c WELCOME TO SHELL

### cat Options

**1) Displaying Non-printing Characters (-v)**

* By default, without any option, this command displays only printing ASCII characters of the file.
* -v option can be used to display even non-printing ASCII characters of the file.
* -n option can be used to number the lines of the file.
* This option helps the programmer in debugging programs.

### wc

* This command can be used to get a count of the total number of lines, words, and characters contained in a file.
* Syntax:

wc FILENAME

LINE WORD CHARATCTER FILENAME 3 3 15 P1.c

// contents of P1.c

$ cat P1.c WELCOME TO

UNIX

$ wc P1.c

* The header includes the following attributes:

### LINE

* This represents the total number of lines in the file.

### WORD

* This represents the total number of words in the file (excluding space, tab and newline).

### CHARATCTER

* This represents the total number of characters in the file (including space, tab and newline).

### FILENAME

* This represents the name of the file.
* This command can also accept more than one filename as arguments.
* Example:

$ wc P1.c P2.c

3 3 15 P1.c

3 3 15 P2.c

### wc Options

–l option can be used to count only number of lines

–w option can be used to count only number of words

–c option can be used to count only number of characters

* Example:

$ wc -l P1.c 3 P1.c

$ wc -c P1.c

15 P1.c

### cp

* This command is used to copy a file(s) from one location to another location.
* It creates an exact image of the file on the disk with a different name.
* Syntax:

cp SOURCE\_FILE DESTINATION\_FILE

* This command can be used to copy a file within current working directory.
* Example:

$ cp FILE1 FILE2 // copies contents of FILE1 to FILE2 in current working directory

* Here,
  + - 1. If the destination file doesn‗t exist, it will first be created before copying takes place.
      2. If the destination file exists, it will be overwritten without any warning from the system.

### Case 2:

* This command can also be used to copy a file to the another directory.
* Example:

$ cp FILE1 part2/FILE2

* Here, this copies the file FILE1 from your current working directory to the file FILE2 in the subdirectory "part2".

### Case 3:

* This command can also be used with .(dot) to signify the current directory as the destination.
* Example:

$ cp part2/FILE2 . same as $ cp part2/FILE2 FILE2

* Here, this copies the file FILE2 in the subdirectory "part2" to your current working directory.

### Case 4:

* This command can also accept more than 2 filenames as arguments.
* In this case, the last filename must be a directory.
* Example:

$ cp FILE1 FILE2 FILE3 module // copies 3 files to "module" directory

* Here, "module" directory should already exist because cp cannot create a directory.

### Case 5:

* This command can also be used with metacharcters (\* or ?).
* Example:

cp \*.pdf DIR1 //copies all the files with extensions .pdf to directory DIR1

### cp Options

1. **Interactive Copying (-i)**

* –i option can be used to warn the user before overwriting the destination file.
* Example:

$ cp –i FILE1 FILE2 // copies contents of FILE1 to FILE2 if "Y" is entered

$ cp: overwrite FILE2 (yes/no)? Y

### Recursive Copying (-R)

* –R option can be used to recursively copy an entire directory structure from one location to another location.
* Entire directory including all files in its subdirectories will be copied.
* Example:

cp –R DIR1 DIR2

* If directory DIR2 doesn‗t exist, cp creates it along with the associated subdirectories.

### mv

* This command renames or moves files.

### Case 1:

* This command can be used to rename a file in the current directory.
* Syntax:

mv OLDFILENAME NEWFILENAME

* It doesn't create a copy of the file; it merely renames it.
* No additional space is consumed on disk during renaming.
* Example:

$ mv OLDFILE NEWFILE // renames the OLDFILE by NEWFILE

### Case 2:

* This command can also be used to move a group of files to a directory.
* In this case, the last filename must be a directory.
* Example:

$ mv FILE1 FILE2 FILE3 module // moves 3 files to "module" directory

### rm

* This command can be used to delete a file.
* Syntax:

rm FILENAME

* Example:

// deletes FILE1

// deletes three files

// deletes all files with extensions .pdf in the currant directory

$ rm FILE1

$ rm FILE1 FILE2 FILE3

$ rm \*.pdf

### rm Options

1. **Interactive Deletion (-i)**

* –i option can be used to warn the user before deleting the file.
* Example:

$ rm –i FILE1 FILE2 // delete FILE1 & FILE2 if "Y" is entered rm: remove FILE1 (yes/no)? ? Y

rm: remove FILE2 (yes/no)? ? Y

### Recursive Deletion (-r or -R)

* –R option can be used to recursively delete an entire directory structure.
* Entire directory including all files in its subdirectories will be deleted.
* Example:

$ rm –r DIR1 // delete DIR1 and all its subdirectories & files

$ rm -r \* // deletes all files in the current directory and all its subdirectories.

### Forceful Deletion (-f)

* By default, this command cannot delete a file which is write-protected.
* –f option can be used to delete even the write-protected file.
* Example:

rm -rf \* // deletes all files in the current directory and all its subdirectories.

### od

* This command can be used to display the content of executable file in a ASCII octal form.
* Example:

$ cat P1.obj

abcd efgh // content of file P1.obj

abcd efgh

### od Options

1. **byte (-b)**

* -b option can be used to display octal value of each printable character.
* Each line displays 16 bytes of data in octal, preceded by the offset of the first byte in the line.
* Example:

$ od –b file

offset <----------------- 16 bytes of data in octal >

0000000 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012

0000020 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012

### character (-c)

* -c option can be used to display the printable characters and its corresponding octal value.

$ od –bc file od -bc ofile

0000000 141 142 143 144 040 145 146 147 148 040 040 040 040 040 040 012

a b c d e f g f \n 0000020 141 142 143 144 040 145 146 147 148

040 040 040 040 040 040 012

a b c d e f g f \n

# MODULE 2(CONT.): BASIC FILE ATTRIBUTES

### File Attributes and Permissions and Knowing them

* ls command can be used to obtain a list of all filenames in the current directory.
* -l option can be used to obtain a detailed list of attributes of all files in the current directory.
* For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| $ ls -l  Type & Perm Link Owner | Group | Size | Date & Time | File Name |
| -rwxr-xr-- 1 kumar | metal | 195 | may 10 13:45 | chap01 |
| drwxr-xr-x 2 kumar | metal | 512 | may 09 12:55 | helpdir |

* The header includes the following attributes:

### Type & Perm

* This represents the file-type and the permission given on the file.
  + The first character indicates type of the file as shown below:
  1. hyphen (-) for regular file ii) d for Directory file

iii) l for Symbolic link file iv) b for block special file v) c for character special file

### File Permission

* + The remaining 9 characters indicates permission of the file.
  + There are 3 permissions: read (r), write (w), execute (x).

1. **Read:** Grants the capability to read, i.e., view the contents of the file.
2. **Write:** Grants the capability to modify, or remove the content of the file.
3. **Execute:** User with execute permissions can run a file as a program.
   * There are 3 types of users: owner, groups and others.
   * The permission is broken into group of 3 characters:
4. The first 3 characters (2-4) represent the permissions for the file's owner.
5. The middle 3 characters (5-7) represent the permissions for the group to which the file belongs.
6. The last 3 characters (8-10) represents the permissions for everyone else.
   * Consider an example -rwxr-xr--
7. Owner has read (r), write (w) and execute (x) permission.
8. Group members have read (r) and execute (x) permission, but no write permission.
9. Others have read (r) only permission.

### Link

* This indicates the number of file names maintained by the system.
* This does not mean that there are so many copies of the file. (Link similar to pointer).

### Owner

* This represents the owner of the file. This is the user who created this file.
* This represents the group of the owner.
* Each group member can access the file depending on the permission assigned.
* The privileges of the group are set by the owner of the file and not by the group members.
* When the system admin creates a user account, he has to assign these parameters to the user:
  1. he user-id (UID) and ii) The group-id (GID)
* This represents the file size in bytes.
* It is the number of character in the file rather than the actual size occupied on disk.

### Date & Time

* This represents the last modification date and the time of the file.
* If you change only the permissions /ownership of the file, the modification time remains unchanged.
* If at least one character is added/removed from the file then this field will be updated.

### File name

* This represents the file or the directory name.

### ls

* This command can be used to list the files and directories stored in the current directory.
* Syntax:

ls [options] [argument]

* For example:

$ ls

bin lib users work

* With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

### ls Options

* -l option can be used to get more information about the listed files.
* When no argument is used, the listing will be of the current directory.
* There are many very useful options for the ls command.
* A listing of many of them follows.
* When using the command, string the desired options together preceded by "-".

|  |  |
| --- | --- |
| a | Lists all files, including those beginning with a dot (.). |
| d | Lists only names of directories, not the files in the directory |
| FInd | icates type of entry with a trailing symbol:   1. executables with \* 2. directories with / and 3. symbolic links ith @ |
|  |  |
| u | Sorts filenames by last access time |
| t | Sorts filenames by last modification time |
| iDis | plays inode number |
| lLon | g listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (→)  precedes the pathname of the linked to file. |

* The mode field is given by the -l option and consists of 10 characters.
* The first character is one of the following:

|  |  |
| --- | --- |
| **Character** | **If entry is a** |
| d | directory |
| - | plain file |
| b | block-type special file |
| c | character-type special file |
| l | symbolic link |
| s | socket |

* The next 9 characters are in 3 sets of 3 characters each.
* They indicate the file access permissions:
  + - 1. the first 3 characters refer to the permissions for the user
      2. the next three for the users in the Unix group assigned to the file, and
      3. the last 3 to the permissions for other users on the system.
* Designations are as follows: r read permission

w write permission

x execute permission

- no permission

* Example:

$ ls -al // To get a long listing of all files in a directory total 24

drwxr-sr-x 5 workshop acs 512 J n 7 11:12 .

drwxr-xr-x 6 root sys 512 May 29 09:59 ..

-rwxr-xr-x 1 workshop acs 532 May 20 15:31 .cshrc

-rwxr-xr-x 1 workshop acs 238 May 14 09:44 .login

-rw-r--r-- 1 workshop acs 273 May 22 23:53 .plan

### Listing Directory Attributes

* $ls -d : can be used to list only names of directories, not the files in the directory.
* However, this command will not list all subdirectories in the current directory.
* For example:

$ls –ld helpdir progs

drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir

drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

* Directories are easily identified in the listing by the first character(d) of the first column.

### Changing File Permissions

* A file is created with a default set of permission.
* chmod command can be used to change permission of a file.
* This command can be used in two ways: 1) Relative mode and 2) Absolute mode.

### Relative Permissions

* This command can be used to add/delete permission for specific type of user (owner, group or others).
* This command can be used to

→ change only those permissions specified in the command line and

→ leave the other permissions unchanged.

* Syntax:

chmod category operation permission filename

* This command takes 4 arguments:
  + - 1. category can be

u → user (owner)

g → group o → others a → all (ugo)

* + - 1. operation can be

+ → assign -

→ remove =

→ absolute

* + - 1. permission can be

r → read w → write

x → execute

* + - 1. Filename whose permission has to changed
* Example:

$ ls -l xstart

-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart

$ chmod u+x xstart // user (u) is added(+) an execute(x) permission

$ ls -l xstart

-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart

$ chmod ugo+x xstart OR chmod a+x xstart // all(a) are added(+) an execute(x) permission

$ ls –l xstart

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

$ chmod go-r xstart // group(g) & others(o) are removed(-) a read(r) permission

$ls –l xstart

-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart

* This command can also accept multiple file names.
* Example:

$ chmod u+x note1 note2 note3

### Recursively Changing File Permissions

* chmod command can be used to change recursively permission of all the files and subdirectories found in the current directory.
* Example:

chmod -R a+x c\_progs // current directory c\_progs

* Here, all files and subdirectories are made executable for all users in current directory c\_progs.

### Absolute Permissions

* This command can be used to add/delete permission for all type of users (owner, group or others).
* This command can be used to change all permissions specified in the command line.
* Syntax:

chmod octal\_value filename

* This command takes 2 arguments:
  + - 1. octal\_value contains 3 octal digits to represent 3 type of users (owner, group or others).
         1. First digit is for user
         2. Second digit is for group and
         3. Third digit is for others

Each digit represents a permission as shown below: 4 (100) – read only

2 (010) – write only

1 (001) - execute only

6 (110) – read & write only

For ex: octal value of 644(110 100 100) means

→ user can read & write only

→ group can read only

→ others can read only

* + - 1. Filename whose permission has to changed.
* Example:

$ ls -l xstart

-rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart // current permission 644

$ chmod 744 xstart same as $ chmod u+x xstart

// user(u) is added(+) an execute(x) permission

$ ls –l xstart

-rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart

$ chmod 755 xstart same as $ chmod a+x xstart

// all(a) are added(+) an execute(x) permission

$ ls –l xstart

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

$ chmod 711 xstart same as $ chmod go-r xstart

// group(g) & others(o) are removed(-) a read(r) permission

$ls –l xstart

-rwx—x--x 1 kumar metal 1906 sep 23:38 xstart

* It is the directory permissions that determine whether a file can be deleted or not.

1. 777 signify all permissions for all categories, but still we can prevent a file from being deleted.
2. 000 signifies absence of all permissions for all categories, but still we can delete a file.

* The system administrator can change the file permissions of every user.

1. Only owner can change the file permissions.
2. User cannot change other user‗s file‗s permissions.

|  |  |
| --- | --- |
| **Relative Permissions** | **Absolute Permissions** |
| $ ls -l xstart | $ ls -l xstart |
| -rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart | -rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart |
| $ chmod u+x xstart | $ chmod 744 xstart |
| $ ls -l xstart | $ ls -l xstart |
| -rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart | -rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart |
| $ chmod ugo+x xstart OR chmod a+x xstart | $ chmod 755 xstart OR chmod a+x xstart |
| $ ls –l xstart | $ ls –l xstart |
| -rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart | -rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart |
| $ chmod go-r xstart | $ chmod 711 xstart |
| $ls –l xstart | $ls –l xstart |
| -rwx—x--x 1 kumar metal 1906 sep 23:38 xstart | -rwx—x--x 1 kumar metal 1906 sep 23:38 xstart |

### Changing File Ownership

* While creating an account on Unix, it assigns a owner ID and a group ID to each user.
* The permissions are assigned to each user based on the Owner and the Groups.
* Two commands are available to change the owner and the group of files:
  + - 1. chown (change owner) is used to change the owner of a file.
      2. chgrp (change group) is used to change the group of a file.

### chown

* This command can be used to change the ownership of a file.
* Syntax:

chown USERNAME FILENAME

* The value of the user can be the name or uid(user id) of a user on the system.
* Example:

$ ls -l note

-rwxr x 1 kumar ISE 347 may 10 20:30 note

$ chown sharma note //now sharma becomes new owner of file "note" ls -l note

-rwxr x 1 sharma ISE 347 may 10 20:30 note

* Now, new owner will have same file permissions as that of old owner.
* Old owner cannot edit "note" since there is no write privilege for group and others.
* Old owner cannot get back the ownership.
* Only super user can change the ownership of any file.
* But normal users can change the ownership of only those files that they own.

### chgrp

* This command can be used to change the group-ownership of a file.
* Syntax:

chgrp GROUPNAME FILENAME

* The value of the group can be the name or GID(group ID) of a group on the system.
* Example:

$ ls -l note

-rwxr x 1 kumar ISE 347 may 10 20:30 note

$ chgrp CSE note //now CSE becomes new group-owner of file "note" ls -l note

-rwxr x 1 kumar CSE 347 may 10 20:30 note

* No super user permission is required.

### Directory Permissions

* The default permissions of a directory are: rwxr-xr-x (755).
  + - 1. Read directory permission grants the ability to view a file.
      2. Write directory permission grants the ability to add, change or remove files from the directory.
      3. Execute directory permission grants the ability to list (ls) the directory content or search (find) for files in the directory.
* Example:

$mkdir c progs

$ls –ld c progs

drwxr-xr-x 2 kumar metal 512 may 9 09:57 c progs

* A directory must never be writable by group and others .
  1. Shells Interpretive Cycle
  2. Wild Cards and Filename Generation
     1. Metacharacters \* and ?
     2. Character Class
  3. Removing the Special Meanings of Wild Cards
  4. Redirection : hree Standard Files
     1. Standard Input
     2. Standard Output
     3. Standard Error
     4. Filters: Using Both Standard Input and Standard Output
  5. Pipe : Connecting Commands
  6. tee : Splitting the Output
  7. Command Substitution
  8. grep
     1. grep Options
  9. Basic Regular Expression (BRE)
     1. Character Class
     2. Asterisk (\*)
     3. Dot (.)
     4. Specifying Pattern Locations (^ and $)
  10. Extended Regular Expression (ERE) and egrep

# MODULE 2 (CONT.): THE SHELL

### Shells Interpretive Cycle

* The shell performs following activities in its interpretive-cycle:
  + 1. The shell

→ issues the prompt ($) and

→ waits for user to enter a command (like ls chap\*).

* + 1. After a command is entered, the shell

→ scans the command-line for metacharacters (like 'ls chap\*') and

→ expands the abbreviations to recreate a simplified command-line ('ls chap1 chap2').

* + 1. Then, the shell

→ passes the command-line to the kernel for execution and

→ waits for the command to complete its task.

* + 1. After the command is executed, the shell

→ issues the prompt ($) again and

→ waits for the user to enter a next command.

### Wild Cards and Filename Generation

* The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

|  |  |
| --- | --- |
| **Wild Card/ Character class** | **Match** |
| \* | Any number of characters including none |
| ? | A single character |
| [ijk] | A single character either an i, j or k |
| [x-z] | A single character that is within the ASCII range of the characters x and z |
| [!ijk] | A single character that is not an i, j, or k |
| [!x – z] | A single character that is not within the ASCII range of the characters x and z |

### Metacharacters \* and ?

* The metacharacter "\*" matches any number of characters including none.
* Examples:

$ ls chap\* // To list all files that begin with "chap" chap chap01 chap02 chap03 chap04 chapx chapy chapz

* When the shell encounters this command line, it identifies the \* immediately as a wild-card. It then looks in the current directory and recreates the command line as below

$ ls chap chap01 chap02 chap03 chap04 chapx chapy chapz

* The shell now hands over this command to the kernel which uses its process creation facilities to run the command.
* The metacharacter ? matches a single character.
* Example:

chap01 chap02 chap03

// To list six-characters filenames beginning with "chap"

// To list all five-characters filenames beginning with "chap"

$ ls chap?

chapx chapy chapz

$ ls chap??

* Both \* and ? operate with some restrictions.
* The \* doesn‟t match all filenames beginning with a dot (.) or forward slash(/).
* Example:

$ ls .C\* // to list all C extension filenames

$ cd /usr?local // this doesn't match /usr/local

### Character Class

* The character class comprises a set of characters enclosed by the rectangular brackets, [ and ],.
* The character class matches a single character in the class.
* For example:

|  |  |
| --- | --- |
| **Character Class** | **Match** |
| [ijk] | A single character either an i, j or k |
| [x-z] | A single character that is within the ASCII range of the characters x  and z. |
| chap0[124] | chap01, chap02, chap04 |
| [!ijk] | A single character that is not an i, j, or k |
| [!x – z] | A single character that is not within the ASCII range of the  characters x and z |

* This can be combined with any string or another wild-card expression.
* Example:

$ ls chap0[124] //Matches chap01, chap02, chap04 and lists if found.

$ ls chap[x-z] //Matches chapx, chapy, chapz and lists if found.

### Negating the Character Class (!)

* Not operator (!) can be used to negate the character class.
* For example,

$ls [!a-zA-Z]\* //To match all filenames that don‟t begin with an alphabetic character.

### Matching Totally Dissimilar Patterns

$ cp $HOME/prog sources/\*.{c,java} . // To copy all the C and Java source programs from

//another directory to the current directory

$ cp /home/srm/{project,html,scripts/\* . // To copy all files from 3 directories (project, //

html and scripts) to the current directory.

* 1. **Removing the Special Meanings of Wild Cards (Escaping and Quoting)**
* Escaping is providing a \ (backslash) before the wildcard to remove its special meaning.
* Example:

$ rm chap\\* // to remove file named "chap\*" "\" is used to suppress special meaning of \*

$ cat chap0\[1-3\] // to list the contents of the file named "chap0[1-3]"

$ rm My\ Document.doc // to remove file named "My Documend.doc"

$ echo \\ // outputs \

* Quoting is enclosing the wild-card within quotes to remove its special meaning.
* When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.
* Example:

$ rm „chap\*‟ // to remove file named "chap\*"

$ rm ―My Document.doc‖ // to remove file named "My Documend.doc"

$ echo "\" // outputs "\"

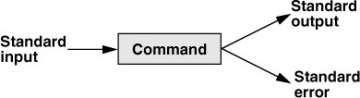
### Redirection : Three Standard Files

* The shell associates three standard files with the terminal:

→ two for display and

→ one for the keyboard.

* When a user logs in, the shell makes available three standard files.
* Each standard file is associated with a default device:
  + 1. Standard input: The file representing input which is connected to the keyboard.
    2. Standard output: The file representing output which is connected to the display.
    3. Standard error: The file representing error messages that come from the command or shell. This file is also connected to the display.



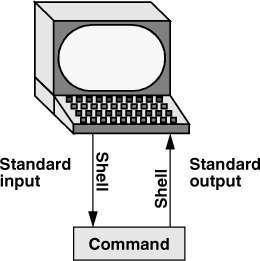


Figure 3.10: Default three Standard Files

### Standard Input

* The standard input can represent three input sources:
  + - 1. The keyboard, the default source.
      2. A file using redirection with the < symbol.
      3. Another program using a pipeline.
* By default, the shell directs standard input from the keyboard.
* Example:

$wc hello world

<ctrl+d> // end of input

2 2 10 // output

* The redirect input symbol (<) instructs the shell to redirect a command‟s input to come from the specified file instead of from the keyboard.
* Example:

$ cat sample.txt hello

world

$ wc < sample.txt

2 2 10 // output

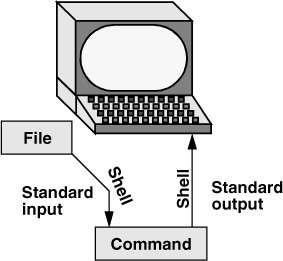


Figure 3.10.1 : Redirecting input

### Standard Output

* The standard output can represent three possible destinations:
  + - 1. The terminal, the default destination.
      2. A file using the redirection symbols > and >>.
      3. As input to another program using a pipeline.
* By default, the shell directs standard output from a command to the screen.
* Example:

$ cat sample.txt hello

world

$ wc sample.txt

2 2 10 // output

* The redirect output symbol (>) instructs the shell to redirect the output of a command to the specified file instead of to the screen.
* Example:

$ wc sample.txt > temp.txt

$ cat temp.txt

2 2 10 // output of wc stored in temp.txt

* >> can be used to append to a existing file.

### Standard Error

* By default, the shell directs standard error from a command to the screen.
* Example:

$ cat empty.txt

cat: cannot open empty.txt // error because empty.txt is non existent file

* The redirect output symbol (>) instructs the shell to redirect the error messages of a command to the specified file instead of to the screen.
* Example:

$ cat empty.txt >> errorfile.txt

$ cat errorfile.txt

cat: cannot open empty.txt // error message of cat stored in errorfile.txt

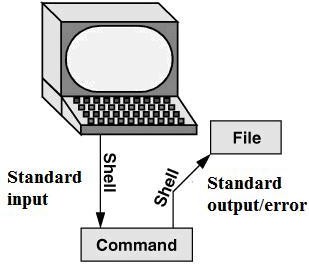


Figure 3.10.2 : Redirecting Output & Error

### Filters: Using Both Standard Input and Standard Output

* UNIX commands can be grouped into four categories:
  + - 1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
      2. Commands like ls, pwd, who etc. don‟t read standard input but they write to standard output.
      3. Commands like lp that read standard input but don‟t write to standard output.
      4. Commands like cat, wc, cmp etc. that use both standard input and standard output.
* Commands like cat, wc, cmp etc. that use both standard input and standard output are called filters.
* Example:

$ cat calc.txt (4\*2)+2

$ bc < calc.txt > result.txt

$ cat result.txt 10

* Here, this command performs arithmetic calculations that are specified as expressions in input file "calc.txt" and redirect the output to a file "result.txt".
* Running 2 or more commands have following disadvantages:

1. The process is slow. The second command cant act unless the first has completed its job.
2. You require an intermediate file that has to be removed after the first command has completed its run.
3. When handling large files, temporary files can build up early and eat up disk space in no time

### Pipe : Connecting Commands

* Pipe is another form of output redirection.
* With piping, the output of a command can be used as input (piped) to a subsequent command.
* Syntax:

$ command1 | command2

* Here, Output from command1 is piped into input for command2.
* The symbol '|' denotes a pipe.
* Example:

$ ls -l | lp

* Here pipe is used to send the output of ls to the lp to print a hard copy of the listing of the current directory.
* Pipeline can be used with 2 or more commands. But the user should know the behavioral properties of each commands to place them there.
* Example:

$ls -l | wc -l | lp // to print a count of files in current directory

### When a Command Needs to be Ignorant of its Source

* If we wish to find total size of all C programs contained in the working directory, we can use the command:

$ wc –c \*.c

* However, it also shows the usage for each file(size of each file). We are not interested in individual statistics.
* Solution: We must make wc ignorant of its input source. So, feed the concatenated output stream of all the .c files to wc –c as its input:

$ cat \*.c | wc –c

### tee : Splitting the Output

* This is an external command that handles a character stream by duplicating its input.
* This command

→ saves one copy in a file and

→ writes the other copy to standard output.

* This command is also a filter and hence can be placed anywhere in a pipeline.
* Example:

$ who | tee users.lst //to display the output of who and save this output in a file users.lst

### Command Substitution

* Command substitution allows to substitute the output of one command into a given command line.
* Syntax:

`command`

* Here, the command must be enclosed between backquotes.
* Example:

$ echo Current date and time is `date`

Current date and time is Sat Nov 24 10:12 IST 2017

$ echo ―There are `ls | wc –l` files in the current directory‖ There are 3 files in the current directory

# MODULE 2 (CONT.): FILTERS USING REGULAR EXPRESSION

### grep

* g/re/p means ―globally search for a regular expression and print all lines containing it‖.
* This command can be used to search a file(s) for lines that have a certain pattern.
* This command

→ scans the file for a pattern and

→ displays

1. lines containing the pattern or
2. line numbers

* Syntax:

grep pattern file(s)

* Example:

grep ―MH‖ student.lst // display lines containing "MH" from the file student.lst

* Patterns with and without quotes is possible.
* Quote is mandatory when pattern involves more than one word.
* Example:

grep ―My Document‖ student.lst // display lines containing "My Document" from student.lst

* This command can be used with multiple filenames.
* Example:

grep ―MH‖ student.lst vtu.lst rank.lst

### 3.14.1 grep Options

* Linux supports below listed options:

|  |  |
| --- | --- |
| -i | ignores case for matching |
| -v | doesn‟t display lines matching expression |
| -n | displays line numbers along with lines |
| -c | displays count of number of occurrences |
| -l | displays list of filenames only |
| -e | Exp specifies expression with this option |
| -x | matches pattern with entire line |
| -f | file takes patterns from file, one per line |
| -E | treats pattern as an extended RE |
| -F | matches multiple fixed strings |

* To understand the working of different options, let us consider we have following information in file student.lst.

|  |  |  |  |
| --- | --- | --- | --- |
| $ ca  4 | t student.  | MH | lst  | 10 | IS | | 111 |
| 4 | | MH | | 11 | CS | | 401 |
| 4 | | GW | | 11 | CS | | 402 |
| 4 | | VV | | 11 | CS | | 403 |

### Ignores Case

* -i (ignore) option can be used to search all lines containing a pattern regardless of uppercase and lowercase distinction
* Example:

$ grep -i "MH" demo file // matches all the words such as MH mH Mh mh 4 | MH | 10 | IS | 111

4 | MH | 11 | CS | 401

### Deleting Lines

* -v option can be used to print all lines that do not contain the specified pattern in a file.
* Example:

|  |  |  |
| --- | --- | --- |
| grep  4 | -v „MH‟ student.lst  | GW | 11 | CS | | 402 |
| 4 | | VV | 11 | CS | | 403 |

### Displaying Line Number

* -n (line number) option can be used to display line numbers containing the pattern.
* Example:

grep -n „MH‟ student.lst 1

2

### Counting Lines Containing the Pattern

* -c (line count) option can be used to count number of lines containing the pattern.
* Example:

grep -c „MH‟ student.lst 2

### Displaying Filenames

* -l (list filename) option can be used to list out the files containing the pattern.
* Example:

grep -l „MH‟ \*.lst

### Matching Multiple Pattern

* -e option can be used to match multiple pattern in a file.
* Example:

grep -e „MH‟ -e „VV‟ student.lst

### Taking Pattern From File

* -f option can be used to place all matched pattern in a separate file, one pattern per line.

### Basic Regular Expression (BRE)

* grep uses an expression of a different type to match a group of similar patterns.
* This command

→ uses an elaborate metacharacter set and

→ can perform amazing matches.

* If an expression uses metacharacters, it is termed a regular expression.
* Regular expression can be classified as
  + 1. BRE (Basic Regular Expression) and
    2. ERE (Extended Regular Expression)
* grep supports

→ BRE by default and

→ ERE with the –E option.

* Character subset of BRE is listed below:

|  |  |
| --- | --- |
| **Metacharcter/ Character Class** | **Match** |
| \* | Zero or more occurrences |
| g\* | nothing or g, gg, ggg, etc |
| . | A single character |
| .\* | nothing or any number of characters |
| [pqr] | a single character p, q or r |
| [c1-c2] | a single character within the ASCII range represented by c1 and c2 |
| ^[pqr] | a single character which is not p, q or r |
| ^Pattern | Pattern at beginning of line |
| Pattern$ | Pattern at end of line |
| ^$ | Line containing nothing |

### Character Class

* The character class comprises a set of characters enclosed by the rectangular brackets, [ and ].
* The character class matches a single character in the class.
* When you use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right.
* caret (^) can be used to negate the character class.
* For example:

|  |  |
| --- | --- |
| **Character Class** | **Match** |
| [pqr] | a single character p, q or r |
| [x-z] | A single character between characters x to z. |
| ^[pqr] | a single character which is not p, q or r |

* Example:

$ grep -i "[Mm][Hh]" demo file // matches all the words such as MH mH Mh mh

### Asterisk (\*)

* The asterisk (**\***) refers to the immediately preceding character.
* It indicates zero or more occurrences of the previous character. g\* → nothing or g, gg, ggg, etc.

lg\* → l or lg, lgg, lggg, etc.

* Example:

$ grep "isaa\*c" demo file // matches isac isaac or isaaac

### Dot (.)

* A dot (.) matches a single character.
* Example:

$ grep "2..." demo file // matches all 4 character words beginning with 2

* Regular expression "**.\*"** → signifies any number of characters or none
* Example:

$ grep prog.c.\* demo\_file // matches all c and cpp extension filenames

### Specifying Pattern Locations (^ and $)

* Following two metacharcters can match a pattern at the beginning or end of a line.

|  |  |
| --- | --- |
| **Metacharcter** | **Match** |
| ^Pattern | Pattern at beginning of line |
| Pattern$ | Pattern at end of line |

* Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurrence only at a particular location.
* Example:

grep ―^2‖ emp.lst //Selects lines starting with 2

grep ―7…$‖ emp.lst //Selects lines where salary between number b/w 7000 to 7999

### Extended Regular Expression (ERE) and egrep

* ERE can be used to match dissimilar patterns with a single expression.
* This uses some additional characters as listed below:

|  |  |
| --- | --- |
| **Metacharcter/**  **Character class** | **Match** |
| ch+ | one or more occurrences of character ch |
| ch? | zero or one occurrence of character ch |
| exp1|exp2 | exp1 or exp2 |
| (x1|x2)x3 | x1x3 or x2x3 |

* If current version of grep doesn‟t support ERE, then use egrep but without the –E option.
* -E option treats pattern as an ERE.
* Example:

$ grep -E "isaa\*c" demo file // matches isac isaac or issaac

$ grep –E „vijaykumar | jayakumar‟ demo file // matches multiple patterns

$ grep –E „(vijay | jaya) kumar‟ demo file // matches multiple patterns

# MODULE 2: SHELL PROGRAMMING

**MORE FILE ATTRIBUTES SIMPLE FILTERS**

* 1. Shell Programming
  2. Ordinary and Environment Variables
     1. Environment Variable
     2. Ordinary(or Local) Variable
  3. File .profile
  4. read and readonly Commands
     1. read Command
     2. readonly Command
  5. Command Line Arguments
  6. exit and Exit Status of a Command
  7. Logical Operators for Conditional Execution
  8. test Command and its Shortcut
     1. Numeric Comparison
     2. String Comparison
     3. File Tests
  9. if Statement
  10. case Statement
      1. Matching Multiple Patterns
      2. Wild-Cards
  11. expr: Evaluate an Expression
  12. while Statement
  13. for Statement
      1. Possible Sources of List
  14. set and shift Commands and Handling Positional Parameters
      1. set
      2. shift
      3. Set -- : Helps Command Substitution
  15. here ( << ) document
  16. trap
  17. Simple Shell Program Examples

# MODULE 2: SHELL PROGRAMMING

### Shell Programming

* A shell script contains a list of commands which have to be executed regularly.
* Shell script is also known as shell program.
* The user can execute the shell script itself to execute commands in it.
* A shell script runs in interpretive mode. i.e. the entire script is compiled internally in memory and then executed.
* Hence, shell scripts run slower than the high-level language programs.
* ".sh" is used as an extension for shell scripts.
* Example: A shell script (program1.sh) to execute few commands. #! /bin/sh

echo ―Welcome to Shell Programming‖ # print message

echo ―Today‘s date : `date`‖ # print date

echo ―My Shell :$SHELL‖ # print shell name

* The hash symbol # indicates the comments in the script.
* The shell ignores all the characters that follow the # symbol. However, this does not apply to the first line.
* The first line

"#! /bin/sh" indicates the path where the shell script is available.

* There are 2 ways to execute a shell script:

### Execute Shell Script Using File Name

* By default, script is not executable. So, the chmod command can be used to make the script executable.
* The scripts are executed in a separate child shell process.
* The child shell reads and executes each statement in interpretive mode. Run:

$ chmod +x program1.sh // add executable permission

$ program1.sh // execute the script program1.sh Output:

Welcome to Shell Programming Today‘s date: Mon Nov 4 11:02:45 IST 2017 My Shell: /bin/sh

### Execute Shell Script by Specifying the Interpreter

* The user can also execute a shell script by specifying the interpreter in the command line.
* Here, the script neither requires a executable permission nor an interpreter line. Run:

$ sh program1.sh //Execute using sh interpreter

$ bash program1.sh //Execute using bash interpreter Output:

Welcome to Shell Programming

oday‘s date: Mon Nov 4 11:02:45 IST 2017

My Shell: /bin/sh

### Ordinary and Environment Variables

* Shell variables are of 2 types: 1) Environment and 2) Ordinary

### Environment Variable

* Environmental variables are used to provide information to the programs you use.
* These variables control the behavior of the system.
* They determine the environment in which the user works.
* If environment variables are not set properly, the users may not be able to use some commands.
* Environment variables are so called because they are available in the user's total environment

i.e. the sub-shells that run shell scripts and mail commands and editors.

* Some variables are set by the system, others by the users, others by the shell programs.
* env command can be used to display environment variables.
* For example:

$ env HOME=home/kumar IFS=' '

LOGNAME=kumar MAIL= /var/mail/kumar MAILCHECK=60

PATH=/bin:/usr/bin PS1='$'

PS2='>'

SHELL=/usr/bin/bash TERM= tty1

### HOME

* This variable indicates the home directory of the current user.
* This variable is set for a user by the system admin in /etc/passwd.

### IFS

* This variable contains a string of characters that are used as word separator in the command line.
* The string normally consists of the space, tab and newline characters.

### LOGNAME

* This variable shows the username.

### MAIL

* This variable specifies the path to user‗s mailbox.

### MAILCHECK

* This variable determines how often the shell checks the file for the arrival of new mail.

1. **PATH**

* This variable specifies the locations in which the shell should look for commands.
* Usually, the PATH variable can be set as follows:

$PATH=/bin:/usr/bin

* The shell has 2 prompts:
  1. The primary prompt $ is the one the user normally sees on the monitor. $ is stored in PS1.  The user can change the primary prompt as follows:

$ PS1="C>"

C> //similar to windows

* 1. he secondary prompt > is stored in PS2.

### SHELL

* This variable specifies the current shell being used by the users.
* Different types of shells are:
  1. Bourne shell /bin/sh 2) C-shell /bin/csh 3) Korn shell /bin/ksh
* This variable is set for a user by the system admin in /etc/passwd.

### TERM

* This variable indicates the terminal type that is used.
* Every terminal has certain characteristics that are defined in a separate control file in the terminfo directory.
* If TERM is not set correctly, vi will not work and the display will be faulty.

### Ordinary(or Local) Variable

* A variable is a character string to which the user assigns a value.
* The value assigned can be a number, text, filename, device, or any other type of data.
* Syntax:

variable = value // variable definition

* The value of variables are stored in the ASCII format.
* For example:

$ x=50

$ echo $x //displays 50

* In command line, all words that are preceded by a $ are identified and evaluated as variables.
* A variable can be removed with unset and protected from reassignment by readonly. Both are shell internal commands.

$ set count=5

$ readonly size = 10

* The variables exist only for a short time during the execution of a shell script.
* The variables are local to the user‗s shell environment.
* The variables are not available for the other scripts or processes.
* As the variables are defined and used by specific users, they are also called user-defined variables.

### Uses of Local variables

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable. Example: $ base=foo ; ext=.c

$ file=$base$ext

$ echo $file // prints foo.c

### File .profile

* A profile file is a start-up file of an UNIX user.
* This file gets executed as soon as the user logs in.
* This file is a shell script that will be present in the home directory of each user.
* The system admin provides each user with a profile with a minimum working environment.
* However, the user can customize the profile as per their requirement.

i.e. The user can

→ assign suitable values to the environment variables.

→ add and modify statements in the profile file.

* This file can be any one of the two:

1. A specific file for each individual user with responsibility for the user environment.
2. A universal file for all users with responsibility for the general environment.

* The user can view his ".profile" as follows:

$ cat .profile

MAIL= /var/mail/kumar PATH=/bin:/usr/bin PS1='$'

PS2='>'

SHELL=/usr/bin/bash ERM= tty1

### read and readonly Commands

* + 1. **read Command**
* read command can be used for taking input from the keyboard.
* It is shell‗s internal tool for making scripts interactive.
* Syntax:

read var\_name

* It is used with one or more variables.
* The variables are used to hold inputs given with the standard input.
* Example: A shell script (program4.sh) to read a search string and filename from the terminal. #!/bin/bash

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Run:

$ program4.sh Output:

What is your name? RAMA Hello, RAMA

### readonly Command

* readonly command can be used to make variables readonly i.e. the user cannot change the value of variables.
* During shell scripting, we may need a few variables, which cannot be modified.
* This may be needed for security reasons.
* Syntax:

variable=value

## For example:

$ readonly PI=3.14

$ echo $PI

$ PI=6.12

//displays 3.14

// this will result in error

### Command Line Arguments

* Shell scripts can accept arguments from the command line.
* ,'. Shell scripts can be run non-interactively and be used with redirection and pipelines.
* The arguments are assigned to special shell variables called shell parameters.
* The shell parameters are reserved for specific functions.
* Different shell parameters:

1. $#: Stores the number of command-line arguments.
2. $0, $1, $2, $3: These are called positional parameters which represent command line arguments.

$0: Stores the filename of the current script.

$1: Stores the first argument.

$2: Stores the second argument

$3: Stores the third argument

1. $\*: Stores all the arguments entered on the command line ($1 $2 ...).
2. "$@": Stores all arguments entered on the command line, individually quoted ("$1" "$2")
3. $?: Stores the exit status of the last command that was executed.
4. $$: Stores Pid of the current shell.
5. $!: Stores PID of the last background job.

* Example: A shell script (program2.sh) to read and display various shell parameters from the command line.

#!/bin/sh

echo "Total Number of Parameters : $#" echo "File Name: $0"

echo "First Parameter : $1" echo "Second Parameter : $2" echo "Quoted Values: $\*" echo "Quoted Values: $@"

$echo "Exit value: $?"

echo "PID of current shell: $$"

Run:

$ program2.sh "RAJA RAM" "MOHAN ROY" Output:

Total Number of Parameters : 2 File Name : program2.sh

First Parameter : RAJA RAM Second Parameter : M HAN R Y

Quoted Values: RAJA RAM M HAN R Y // stored as "RAJA RAM MOHAN ROY"

Quoted Values: RAJA RAM MOHAN ROY // stored as array= {"RAJA RAM" , "MOHAN ROY"} Exit value: 0 // zero implies success

PID of current shell: 12345

### exit and Exit Status of a Command

* exit command can be used to terminate a program(or script).
* This command returns value which will be available to the script's parent process.
* The $? variable contains exit status of the last command executed.
* Exit status is a numerical value returned by every command upon its completion.
* A command returns an exit status of

1. zero (0) upon successful execution and
2. non-zero upon unsuccessful execution i.e. an error condition.

* Exit status can be used to devise program-logic that branches into different paths depending on success or failure of a command.
* Example: A shell script to find relationship between 2 numbers. #! /bin/usr

x=5; y=7

test $x –eq $y; echo "5=7: $? \n" test $x –ne $y; echo "5!=7: $? \n" test $x –gt $y; echo "5>7: $? \n " test $x –ge $y; echo "5>=7: $? \n " test $x –lt $y; echo "5<7: $? \n " test $x –le $y; echo "5<=7: $? \n"

// Returns nonzero exit status i.e. failure → False

// Returns zero exit status i.e. success → True

// False

// False

// True

// True

Output: 5=7: 1

5!=7: 0

5>7: 1

5>=7: 1

5<7: 0

5<=7: 0

### Logical Operators for Conditional Execution

* Two logical operators can be used for conditional execution: 1) && and 2) ||

### && Operator

* Syntax:

cmd1 && cmd2

* Here, cmd2 gets executed only when cmd1 succeeds.

### || Operator

* Syntax:

cmd1 || cmd2

* Here, cmd2 gets executed only when cmd1 fails.
* Example: A script to illustrate the usage of && and ||.

$ cat student.lst

4 | MH | 10 | IS | 111

4 | MH | 11 | CS | 401

4 | GW | 11 | CS | 402

4 | VV | 11 | CS | 403

$ grep ‗VV‘ student.lst && echo ―Pattern found‖

4 | VV | 11 | CS | 403

Pattern found

$ grep ‗ZZ‘ student.lst || echo ―Pattern not found‖

Pattern not found

### test Command and its Shortcut

* Usually, if-construct cannot directly handle the true or false value returned by evaluation of an expression.
* So, test command can be used to handle the true or false value returned by evaluation of an expression.
* Test command

→ uses certain operators to evaluate the condition on its right and

→ returns either a true or false exit status.

* Then, if-construct uses the exit status for making decisions.
* Test command

→ does not display any output

→ sets the parameter $? (exit status).

* Test command works in 3 ways:

1. Compare two numbers.
2. Compares two strings or a single one for a null value.
3. Checks files attributes.

### Numeric Comparison

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| -eq | Equal to |
| -ne | Not equal to |
| -gt | Greater than |
| -ge | Greater than or equal to |
| -lt | Less than |
| -le | Less than or equal |

* Syntax:

test $op1 -operator $op2

* Operators always begin with a – (Hyphen) followed by a two character word .
* Numeric comparison can be done on integer values only. (The decimal values are truncated).

### Shorthand for test

* [ and ] can be used instead of test.
* Example: A shell script to find relationship between 2 numbers. #! /bin/usr

x=5; y=7

test $x –eq $y; echo "5=7: $? \n" test $x –ne $y; echo "5!=7: $? \n" test $x –gt $y; echo "5>7: $? \n " test $x –ge $y; echo "5>=7: $? \n " test $x –lt $y; echo "5<7: $? \n" test $x –le $y; echo "5<=7: $? "

// False

// True

// False

// False

// True

// True

Output: 5=7: 1

5!=7: 0

5>7: 1

5>=7: 1

5<7: 0

5<=7: 0

### String Comparison

* Test command is also used for testing strings.

|  |  |
| --- | --- |
| **Operator** | **True if** |
| s1=s2 | String s1=s2 |
| s1!=s2 | String s1 is not equal to s2 |
| -n stg | String stg is not a null string |
| -z stg | String stg is a null string |
| stg | String stg is assigned and not null |

* Example: A shell script to check if 2 strings are equal or not. #!/bin/sh

echo ―Enter the first string: \c‖ read str1

if [ -z ―$str1‖ ] ; then

echo ―You have not entered the string‖; exit 1 echo ―Enter the second string: \c‖

read str2

if [ -z ―$str2‖ ] ; then

echo ―You have not entered the string‖; exit 1 if[

$str1= $str2] then

else

echo "Both strings are equal" echo "Strings are unequal"

Output:

Enter the first string: MAM Enter the second string: MAM Both strings are equal

### File Tests

* Test command can be used to check various file attributes such as file type (-, d or l) & file permission (r, w, x).

|  |  |
| --- | --- |
| **Test** | **True if** |
| -e file | File exists |
| -f file | File exists and is a regular file |
| -d file | File exists and is a directory |
| -L file | File exists and is a symbolic link |
| -r file | File exists and readable |
| -w file | File exists and is writable |
| -x file | File exists and is executable |
| -s file | File exists and has a size greater than zero |
| f1 –nt f2 | File f1 is newer than f2 |
| f1 –ot f2 | File f1 is older than f2 |
| f1 –ef f2 | File f1 is linked to f2 |

* Example: A shell script (program8.sh) to check whether a file has permission for read, write and execute.

#! /bin/usr

echo -n "Enter file name:" read file if [–e $file] ;

then

else fi

echo ―File exists \n‖

echo ―File does not exist \n‖

if [ -r "$file" ] then

else fi

echo "File is readable \n " echo "File is not readable \n "

if [ -w "$file" ] then

else fi

echo "File is writable \n " echo "File is not writable \n "

if [ -x "$file" ] then

else fi

echo "File is executable \n " echo "File is not executable \n "

Run:

$ ls –l student.lst

-rw-rw-rw- 1 kumar group 870 jun 8 15:52 student.lst $ program8.sh

Output:

Enter file name: student.lst File exists

File is readable File is writable

File is not executable

### if Statement

* if statement is basically a ―two-way‖ decision statement.
* This is used when we must choose between two alternatives.
* Three forms of if…else statement:

1. if...fi statement
2. if...else...fi statement
3. if...elif...else...fi statement

* Syntax 1:

if command is successful then

execute statements

fi

* Syntax 2:

if command is successful then

else fi

* Syntax 3:

execute statements execute statements

if command is successful then

execute statements elif command is successful

then

else

fi

execute statements execute statements

* Here is how it works:

1. If the command succeeds, the statements within then-block are executed.
2. If the command fails, the statements within else-block are executed.

* Example: A script to check whether an integer is positive or negative. #! /bin/sh

echo ―Enter any non zero integer: \n‖

read num

if [$num -gt 0]; then

echo ―Number is positive number‖

else

echo ― umber is negative number‖

Output:

Enter any non zero integer: 5

Number is positive number

### case Statement

* case statement is basically a ―multi-way‖ decision statement.
* This is used when we must choose among many alternatives.
* This also handles string tests, but in a more efficient manner than if statement.
* Syntax:

case expression in

pattern1) statement1 ;; pattern2) statement2 ;;

pattern3) statement3 ;;

…

esac

* Here is how it works:

1. Firstly the expression is matched with pattern1.
2. If the match succeeds, then statement1 will be executed.
3. If the match fails, then the expression is matched with pattern2 and this process continues.

* Each statement is terminated with a pair of semicolon (;;).
* This can match only strings but cannot handle numeric and file tests.
* This is very effective when the string is fetched by command substitution.
* Example: A script to display appropriate message based on grades (A to D). #! /bin/sh

echo ―enter grade A to D

\n‖ read grade

case ―$grade‖ in

* 1. echo ―Excellent!‖ ;;
  2. echo ―Well done‖ ;;
  3. echo ―You passed‖ ;;
  4. echo ―Better try again‖ ;;

\*) echo ―Invalid grade‖ ;;

esac

echo ―Your grade is $grade‖

Output:

enter grade A to D B

Well done

Your grade is B

### Matching Multiple Patterns

* case statement can also specify the same action for more than one pattern.
* Example: A script to test a user response for both y and Y (or n and N). #! /bin/sh

echo ―Do you wish to continue? [y/n]:‖

read ans

case ―$ans‖ in

Y | y ) ;;

N | n ) exit ;;

esac

### Wild-Cards

* case statement has a superb string matching feature that uses wild-cards.
* case statement uses

→ filename matching meta-characters \* and ?

→ string matching character.

* Example: A script to test a user response for YES, yes, Yes, yEs (or no, NO, No, nO). #! /bin/sh

echo ―Do you wish to continue? [y/n]:‖

read ans

case ―$ans‖ in

[Yy] [eE]\* ) ;; # Matches YES, yes, Yes, yEs, etc

[Nn] [oO] ) exit ;; # Matches no, NO, No, nO

\* ) echo ―Invalid Response‖

esac

### expr: Evaluate an Expression

* expr command can be used to

→ evaluate an expression and

→ output the corresponding value.

* This command combines the following two functions:

1. Performs arithmetic operations on integers and
2. Manipulates strings.

### Numeric Computation

* Five operators used on integers: +, -, \*, / and %.
* Syntax:

expr $op1 operator $op2

* Example:

$ x=5 y=3

$ expr $x + $y // outputs 8

$ expr $x - $y // outputs 2

$ expr $x \\* $y // \* must be escaped to prevent shell from interpreting \* as wildcard

//outputs 15

$ expr $x / $y //outputs 1

$ expr $x % $y // outputs 2

$ z =`expr $x + $y` // command substitution to assign a variable

$ echo $z // outputs 8

### String Handling

* Three functions used on strings:
  1. Finding length of string
  2. Extracting substring
  3. Locating position of a character in a string
* Syntax:

expr "exp1" : "exp2"

* On the left of the colon (:), the string to be worked upon is placed.

### Length of the String

* The regular expression ".\*" is used to print the number of characters matching the pattern.
* Syntax:

expr "string" : ".\*"

* Example:

$ expr ―vtunotesbysri‖ : ‗.\*‘ // outputs 13

### Extracting a Substring

* expr command can be used to extract a string enclosed by the escape characters "\(" and "\)".
* Syntax:

expr "string" : "\( substring \)"

* Example:

$ expr ―vtunotesbysri‖ : " \( sri \)" // outputs 'sri'

### Locating Position of a Character

* expr command can be used to find the location of the first occurrence of a character inside a string.
* Syntax:

expr "string" : "[^ch]\*ch" //ch → character

* Example:

$ expr ―vtunotesbysri‖ : "[^u]\*u" // outputs 3

### while Statement

* while loop can be used to execute a set of statements repeatedly as long as a given condition is true.
* Syntax:

while condition is true do

execute statements

done

* The statements enclosed between do and done are executed repeatedly as long as condition is true.
* Example: A script to display a message 3 times using while loop. #! /bin/sh

num=1

while [$num -le 3] do

echo " Welcome to Shell Programming " expr $num = $num +1;

done

Output:

Welcome to Shell Programming Welcome to Shell Programming Welcome to Shell Programming

### for Statement

* for loop can be used to iterate over all items(or strings) within a list.
* Syntax:

for variable in list do

statements

done

* Here, list consists of a set of items(or strings).
* Each item of the list is picked up and assigned to the "variable".
* The iteration continues until all items are picked from the array.
* Example: A script to display elements of an array. #! /bin/sh

print("Here are the numbers in the list: \n");

for var in 10 20 30 40 50 60; do

echo ―$var \t‖

done

Output:

Here are the numbers in the list 10 20 30 40 50 60

### Possible Sources of List

* Possible sources of list are
  + - 1. List from variables
      2. List from command substitution
      3. List from wildcards and
      4. List from positional parameters

### List from Variables

* Example: A script to evaluate & display a set of variables using for-loop. #! /bin/sh

x="Dream" y="Believe " z="Achieve"

for var in $x $y $z;

do

echo ―$var \t‖

done

Achieve

Believe

Output: Dream

### List from Command Substitution

* Command substitution can be used for creating a list.
* Useful: when list is large.
* Example: A script to display current date using for-loop. #! /bin/sh

for var in `date`

do

echo ―$var \t‖

done

08:02:45 IST2017

Output:

Mon Nov 4

### List from Wildcards

* The shell can use wildcards for matching filenames.
* Example: A script to print all files with pdf extension. #! /bin/sh

for file in \*.pdf do

echo "Printing $file \n" lp $file

done

Output:

Printing chap1.pdf Printing chap2.pdf Printing chap3.pdf

### List from Positional Parameters

* Example: A script (program4.sh) to read & display a positional parameters using for-loop. #! /bin/sh

for var in "$\*" # even "$@" can be used

do

echo "$var \t"

done

Run:

$ program4.sh A B C Output:

A B C

### set and shift Commands and Handling Positional Parameters

* + 1. **set**
* set command can be used to assign positional parameters ($1, $2 and $3) to command line arguments.
* This command can be used for picking up individual fields from the output of a program.
* Example:

$ set 98 23 62

* Here, above line assigns

→ 98 to $1

→ 23 to $2

→ 62 to $3.

* This command can also be used to assign the other parameters $# and $\*.
* Example:

$ set `date`

$ echo $\*

Mon Nov 4 08:02:45 IST 2017

* Example:

$ echo ―The date today is $2 $3, $6‖

The date today is Nov 4, 2017

### shift

* shift command is a shell built-in that operates on the positional parameters.
* Each time shift command is called, it shifts/transfers all the positional parameters down by one.
* For example: $2 becomes $1

$3 becomes $2

$4 becomes $3, and so on.

* Example:

$ echo ―$@‖

Mon Nov 4 08:02:45 IST 2017

$ echo $1 $2 $3 Mon Nov 4

# $@ and $\* are interchangeable

$ shift

$ echo $1 $2 $3

Nov 4 08:02:45

# shifts 1 place

$shift 2

$echo $1 $2 $3

# shift 2 places

08:02:45 IST 2017

### Set -- : Helps Command Substitution

* Problem with set command:

When set command is used with command substitution, the output of the command may begin with a -(hypen). In this case, set command interprets -(hypen) as an option and does not work correctly.

* For example:

$set 'ls -l student,lst'

-rwxr-xr--: bad option

* Solution: Use --(double hypen) immediately after set command.

$set -- 'ls -l student.lst'

-rwxr-xr-- 2 kumar group 163 Jul 13 21:36 student.lst

### here ( << ) document

* The << symbol can be used to read data from the same file containing the script. This file is called as a here document.
* The term 'here' signifies that the data is here rather than in the file.
* Any command using standard input can also take input from a here document.
* Syntax:

command << delimiter document

delimiter

* For example:

$ mailx kumar << MARK Explore

Dream Discover

MARK

* The string (MARK) is delimiter.
* The shell treats every line delimited by MARK as input to the command mailx.
* kumar at the other end will see 3 lines of message text with the date inserted by command.
* The word MARK itself doesn‗t show up.

### Using Here Document with Interactive Programs:

* A shell script can be made to work non-interactively by supplying inputs through here document.
* For example:

$ wc -l << END

Decide Commit Succeed

END

3 //outputs number of lines = 3

### trap

* trap is a signal handler.
* Whenever the interrupt key (Ctrl+C) is pressed, a signal SIGINT is sent to terminate the shell script.
* However, it is not a good practice. For instance, the user may end up leaving a lot of temporary files on the disk.
* trap command can be used to perform clean up operation when a script receives a terminate signal.
* This command is normally placed at the beginning of the shell script.
* Syntax:

trap command\_list signal\_list

* The signal list contains the signal names (SIGINT, SIGTERM, SIGQUIT).
* The command list contains the commands to be executed when the signals are received by the script.
* Two common uses of trap:

1. Clean up temporary files and
2. Ignore signals

### Cleaning up Temporary Files

* The user can remove some files and then exit if someone tries to abort the script from the terminal.
* Example:

$ trap ‗rm temp.txt ; exit‗ SIGINT

* Here, a file temp.txt will be automatically removed if a signal SIGINT is received by the script.

### Ignoring Signals

* A script can be made to ignore a specific signal by using a null command list.
* Example:

trap ‗ ‗ SIGINT

* Here, the script ignores a signal SIGINT when it is received.

### Simple Shell Program Examples

1. A shell script to accept a filename as argument and displays the last modification time if the file exists and a suitable message if it does not.

#!/bin/bash

echo "Enter name of the file: \c" read filename

if [ -e $filename ] then

else fi

echo 'Last modification time is: \c'

echo `ls -l $filename | cut -d " " -f 6,7,8` echo "file does not exist"

Output:

Enter name of the file: student.lst

Last modification time is: Nov 04 12:04:11

1. A shell script to accept 2 file names & check if the permission for these files are identical and if they are not identical, display each filename followed by permission.

#!/bin/bash

echo "Enter 2 filenames: \c" read f1 f2

file1 =`ls -l $f1 | cut -c 2-10` file2 =`ls -l $f2 | cut -c 2-10` if [ $file1 == $file2 ] then

echo "Common file permission: $file1"

else

fi

echo "Different file permissions " echo " permission of $f1: $file1" echo " permission of $f2: $file2"

Output:

Enter 2 filenames: p1.c p2.c Different file permissions

file permission for p1.c is rw-r--r-- file permission for p2.c is rwxr-xr-x

1. A shell script to print first 10 numbers (1 to 10) #!/bin/sh

x=0

while [$x –le 10]; do

echo ―$x \t‖ x=`expr $x+1`

done

Output:

1 2 3 4 5 6 7 8 9 10

1. A shell script (program4.sh) to accept any number of arguments and print them in a reverse order. For example if A B C are entered then output is C B A.

#!/bin/bash n=$#

if [ $n -lt 2 ]; then

else

echo "please enter 2 or more arguments" exit

echo "The command line arguments in reverse order:" while [ $n -ne 0 ]

do

eval echo "\$$n" #display values in positional parameters $3 $2 $1 n = `expr $n - 1`

done

fi

Run:

$ program4.sh A B C Output:

C B A

1. A shell script to create a menu, which displays the list of files, process status, current date and current users of the system.

#! /bin/sh

echo ― MENU \n

* 1. List of files 2. Processes of user \n

3. Today‘s Date 4. Users of system \n

1. Quit \n

Enter your option: \c‖ read choice

case ―$choice‖ in

* 1. ls –l;;
  2. ps –f ;;
  3. date ;;
  4. who ;;
  5. exit ;;

\*) echo ―Invalid option‖

esac

// date command executed

Output:

MENU

1. List of files 2. Processes of user

3. Today‘s Date 4. Users of system

5. Quit

Enter your option: 3

Mon Oct 8 08:02:45 IST 2007

* 1. A shell script to read a string from terminal and display suitable message if it doesn't have at least 10 characters using

expr. #!/bin/sh

echo ―Enter a string: \c‖ read str

length = `expr ―$str‖ : ".\*" ` if [

$length -lt 10 ] then

else fi

echo "The string has less than 10 characters" echo "The string has $length characters"

Output:

Enter a string: vtunotesbysri The string has 13 characters

* 1. A shell script to read a string from terminal and display suitable message if it doesn't have at least 10 characters using

case. #!/bin/sh

echo ―Enter a string: \c‖ read str

length = (${#str}<10) case $length in

* + 1. echo "The string has less than 10 characters" ''

\*) echo "The string has $length characters" ;;

esac

Output:

Enter a string: vtunotesbysri The string has 13 characters

* 1. A shell script to check whether a given number is palindrome or not #!/bin/sh

echo "Enter the number: \c" read n

number=$n reverse=0 while [ $n -gt 0 ]

do

a=`expr $n % 10 ` n=`expr $n / 10

` reverse=`expr $reverse \\* 10 +

$a`

done echo $reverse

if [ $number -eq $reverse ] then

else fi

echo "Number is palindrome" echo "Number is not palindrome"

Output:

Enter the number: 1221 Number is palindrome

* 1. A shell script to read a pattern and filename from the terminal. And search for the pattern in the file.

#! /bin/sh

echo ―Enter the pattern to be searched: \c‖ read pname

echo ―Enter the file to be used: \c‖ read fname

echo ―Searching for pattern $pname from the file $fname‖ grep

$pname $fname

echo ―Selected records shown above‖

Output:

Enter the pattern to be searched : MH Enter the file to be used: student.lst

Searching for pattern MH from the file student.lst 4 | MH | 10 | IS | 111

4 | MH | 11 | CS | 401

Selected records shown above

* 1. A shell script (program10.sh) to compute sum of numbers passed in command line #!/bin/sh

sum=0

for I in ―$@‖

do

sum =`expr $sum + $I`

done

echo ―sum is $sum‖

Run:

$ program10.sh 2 4 6 Output:

sum is 12

* 1. A shell script (program11.sh) to compute length of strings in the file (student.lst) #!/bin/sh

sum=0

for I in `cat student.lst `; do

echo ―string is $I \n‖ x= `expr ―$I‖:‘.\*‘` echo

―length is $x \n‖

done

Run:

$ cat student.lst RAMA KRISHNA

$ program11.sh Output:

string is RAMA length is 4

string is KRISHNA length is 7

* 1. A shell script to validate the password. Let VALID\_PASSWORD="secret" #!/bin/sh

echo "Please enter the password:" read PASSWORD

if [ "$PASSWORD" == "$VALID\_PASSWORD" ]; then

echo " Login successful"

else fi

echo "ACCESS DENIED!"

Output:

Please enter the password: secret Login successful

* 1. A shell shell script to append doc extension to all filenames. #!/bin/sh

for file in ch1 ch2 ch3;

do

cp $file ${file}.doc

echo $file copied to $file.doc

done

Output:

ch1 copied to ch1.doc ch2 copied to ch2.doc ch3 copied to ch3.doc

* 1. A shell script to check if the length of the name is greater than 20 characters. #!/bin/sh

echo ―Enter your name: \c‖ read name

if [`expr ―$name‖ : ".\*" `-gt 20] ; then echo

―Name is very long‖

else fi

echo ―You can proceed!‖

Output:

Enter your name: Rama Krishna You can proceed!