U.P.Rajarshi Tandon Open University, Prayagraj

Master of Computer Application

MCA-122
Unix and Shell Programming

Course Design Committee

Prof. Ashutosh Gupta — **Chairman**
Director (In-charge)
School of Science, UPRTOU Allahabad

Prof. Suneeta Agarwal — **Member**
Dept. of Computer Science & Engineering
Motilal Nehru National Institute of Technology Allahabad

Dr. UpendraNathTripathi — **Member**
Associate Professor
DeenDayalUpadhyay Gorakhpur University, Gorakhpur

Dr. Ashish Khare — **Member**
Associate Professor
Dept. of Computer Science, University of Allahabad, Prayagraj

Ms. Marisha — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU Allahabad

Mr. Manoj Kumar Balwant — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU Allahabad

Course Preparation Committee

Dr. Chandrakant kumar Singh — **Author**
Assistant Professor, Computer science
School of Sciences, UPRTOU, Prayagraj

Prof. Manu Pratap Singh — **Editor**
Professor, Department of Computer Science Engineering
Dr. BhimraoAmbedkar University, Agra

Mr. Manoj Kumar Balwant — **Coordinator**
Assistant Professor (Computer Science),
School of Sciences, UPRTOU, Prayagraj

First Edition: July 2024

ISBN: -978-93-48270-98-6

Printed and Published by Vinay Kumar Registrar, Uttar Pradesh Rajarshi Tandon Open University 2024.

Printed by : Chandrakala Universal Pvt.Ltd. 42/7 JLN Road, Prayagraj, 211002.

BLOCK-1 INTRODUCTION

This block is a course on "Introduction to Unix Operating System". This block course contains four units. Unit 1 is an introductory unit on UNIX operating system. With this unit, the learners will be acquainted with some important concepts like Unix introduction - Basic Features, advantages, Basic Architecture of Unix system, Kernel, Types of shells: Bourne shell, C Shell, korn shell, unix commands. Unit 2 discusses the Working with Files and Directories. This includes the Unix file system, creating files, Listing Files and Directories, masking file permissions, directory permissions, removing a file forcibly, Mkdir, ls, pwd and cd,echo and cat,wc,ls –l, other useful variations of ls, tput command . Unit 3 concentrates on Unix File system. In this unit, the learners will be acquainted with some important concepts file system like Boot block, super block, Inode table, data blocks, How Unix access files, storage files, Disk related commands: checking disk free space, dfsapce, du, ulimit. Unit 4 is the last unit and it discusses about Essential Commands of Unix.and cover the topic lke Password, cal, banner, touch, file, links with DOS, Commands for files and directories, cd, ls, cp, md, rm, mkdir, rmdir, more, less, creating and viewing files, using cat, file comparisons, file compression commands, unix manual page, setting PATH, System startup and shut-down.

The learners will learn how to work with Unix operating system. The units of this block also discusses abut Unix Operating System, Working with Files and Directories and Essential Commands of Unix While going through a unit, you will notice some boxes along-side, which have been included to help you know some of the difficult, unseen terms. Some "ACTIVITY' (s) have been included to help you apply your own thoughts. Again, we have included some relevant concepts in "LET US KNOW" along with the text. And, at the end of each section, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be better if you solve the given problems in these boxes immediately, after you finish reading the section in which these questions occur and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

UNIT-1 : INTRODUCTION TO UNIX OPERATING SYSTEM

Structure

1. INTRODUCTION

A computer system can be divided into 4 components:

- Hardware (CPU, memory, input/output devices, etc.)
- Operating system,
- System programs (word processors, spread sheets, accounting software‘s, compilers,)
- Application programs.

In the 1960s, an operating system was defined as software that manages hardware. However, with the advent of microcode, this definition has evolved. Today, an operating system is understood as a collection of programs that makes hardware functional and accessible. Essentially, an operating system is the software that governs a computer's operations, establishing connections between the user, software applications, and hardware. It serves as an interface among these components. Every computer requires an operating system (OS) to operate effectively. The OS is the first software loaded when a computer boots up, with all subsequent application programs being loaded afterward.

1A. OBJECTIVES

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of UNIX operating system
- Types of Operating System
- Feature of UNIX
- Basic Architecture of Unix system
- Advantages and Disadvantages of UNIX operating system
- Basic Concept of Kernel and shells
- Type of shell, C

- Basic unix commands

1.1 TYPES OF OPERATING SYSTEM

Single User:

If a single-user operating system is loaded into a computer's memory, the computer will be able to handle only one user at a time. Ex: MS-DOS, MS-Win 95-98, Win-ME

Multi user:

If the multi-user Operating System is loaded in computer‘s memory; the computer would be able to handle more than one user at a time. Ex: UNIX, Linux, XENIX

Network: If the network Operating System is loaded in computer‘s memory; the computer would be able to handle more than one computer at time. Ex: Novel Netware, Win-NT, Win-2000-2003

1.2 WHAT IS UNIX

UNIX is an operating system designed to support multiple users simultaneously. It was developed around 1969 at AT&T Bell Labs by Ken Thompson and Dennis Ritchie.

The UNIX operating system functions as an intermediary between the computer hardware and the user, consisting of a collection of programs.

The core component of UNIX, known as the kernel, manages system resources and handles the computer's internal operations.

Users interact with the kernel through a program called the shell, which serves as a command-line interpreter. The shell translates user commands into a format that the kernel can understand and process.

- **Types of UNIX**

  There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

  The UNIX operating system is made up of three parts; the kernel, the shell and the programs

1.3 FEATURE OF UNIX

Unix is a computer operating system. The salient feature of Unix Operating system are as follows –

1.3.1 Multi User:

Multitasking is the capability of the operating system to perform various tasks. i.e., A single user can perform various tasks.

1.3.2 Multi Tasking:

This allows several users to use the same computer to perform their tasks.

1.3.3 Portability:

UNIX is portable because it is written in a high level language . So UXIX can be run on different computers

1.3.4 Communication:

UNIX supports the following communications.

Between the different terminals connected to the UNIX server.

Between the users of one computer to the users of another

1.3.5 Security:

Every user has a login name and a password. So, accessing another user‘s data is impossible without permission

1.3.6 Programming facility:

UNIX is highly programmable; the UNIX shell programming language has all the necessary ingredients like conditional and control structures (Loops) and variables.

1.3.7 Machine Independent:

It hides the machine architecture from the user. So it is easier to write a program that run on different hardware implementation – Unix Os treats everything including memory and I/O devices as files – The Unix file system allows easy and efficient maintenance of files

1.3.8 Paging:

If the available memory is full. LINUX then look for 4kb pages of memory which can be freed. Pages, with content already on hard disk are discarded. If one of the pages is to be accessed, it has to be reloaded. This procedure is called paging

- **Other features**
  - Distributed data processing capabilities
  - Open Source code
  - Shell Scripts
  - Powerful Pipes and Filters
  - E-Mail Facility
  - Support to most programming languages

1.4 ADVANTAGES OF UNIX OPERATING SYSTEM

1.4.1 Ecommerce:

Many big online stores use UNIX or Linux servers to host their websites. UNIX has also used to manage mobiles and electronic machines.

1.4.2 Portable:

As UNIX is made from using C language so it is a highly portable OS. You can use this OS on any PC or MAC computers. C is a very popular language and most programmers can work easily in this language. You can communicate with hardware by using the C language.

1.4.3 Memory usage:

UNIX use less memory while running sophisticated programs. UNIX OS can handle virtual memory nicely. The virtual memory expands as more programs come into the main memory. Most of the tasks in UNIX is done by using fewer resources.

1.4.4 Less code to execute:

In GUI (graphical user interface), we sometimes need many mouse clicks to perform some specific task but in the case of UNIX we can simply write one command in CLI (command line interface) and that task is done.

1.4.5 Files used everywhere:

All type of data is stored in files i.e. all devices and terminal are stored in files. Working with files in OS becomes fast and can be easily managed by UNIX.

1.4.6 Ideal for web hosting:

As UNIX OS is free and secure so it is widely used by web hosting companies. Many web hosting servers use utilities like DNS (Domain Name Server), DHCP (Dynamic Host Configuration Protocol) and the web server.

1.4.7 Safe and secure:

UNIX provides a safe and secure platform in which multiple users can interact with the servers online without any security issues. The interaction with the UNIX servers is fast and without any bugs. UNIX uses UID and GID for controlling permissions for users and files are accessed by users through these permissions

1.4.8 No anti-virus needed:

As you know that Chrome OS, macOS, Linux, Ubuntu and android are developed by using UNIX OS. These OS are considered safe from any virus. You don't need to install any anti-virus in newly build Chrome OS.

1.4.9 User interactions:

There are many online servers, electric machines where user interaction is not involved. UNIX is an ideal choice for machines and devices where no or fewer user interactions are involved. UNIX can manage the automatic working of systems very well.

1.4.10 supports multiple users:

In UNIX every user needs a username password to use the OS. Every file is protected from unauthorized use. Multiple users can log in to the system and use the OS as they need. You can take the example of RDP (Remote Desktop Protocol) server and VPS (Virtual Private Server). In RDP and VPS multiple users can log in to the system and every login gives private files access to the user. Every user has a user access control system by which they can access the files securely. All the users can open many apps at the same time and there are very few chances that your system may crash. The owner of the system can set the permission level to all the users and then any user can access the files as set by the owner.

1.4.11 String commands and utilities:

If you want to put text inside binary data and tries to fetch the text then it is done by using string commands and utilities. You can combine small commands into complex commands by using string commands. There are more than 400 command and utilities in UNIX by which you can do all type of tasks.

1.4.12 Looks the same as MS-DOS:

If you have experience in using MS-DOS in windows then it becomes easy for you to use UNIX. The usage of commands and user interface is quite similar to MS-DOS.

1.4.13 Multitasking:

You can open many programs in the UNIX OS and all the programs work in parallel using multiprocessor technology.

1.4.14 Modular:

The utilities for UNIX are made in modular form. If you are a programmer then you can make small programs in modular form and then unite the modules and all the modules will work consistently throughout the system.

1.4.15 Human readable source code:

As most of the code is written in C language and is available as open-source so any user can see and understands the code. The source code is written in the English language. I know English words are different from source code but if you understand the basics of programming and also understand English then you can easily follow through with the code and change it.

1.4.16 Many vendors:

UNIX has many popular vendors and standards including POSIX, AIX, and HP-UX.

1.4.17 Hiring experts:

As UNIX is more than 50 years old so there are many expert developers available that can help you to sort out your problem in the OS. The demand for UNIX developers is also high because most cloud-based applications run on UNIX. Online servers and mobile OS like Android is also developed in UNIX.

1.4.18 Many variants available:

There are many types of UNIX variants are available for UNIX OS. If you are not comfortable using Linux then you can use Ubuntu, Redhat or macOS. Every UNIX OS has some type of different UI. You are free to use any type of UNIX OS.

1.4.19 Used in a large organization:

UNIX is used in universities, research laboratories, colleges and large government organizations. Many students and researchers use UNIX for code learning and get expertise in using operating systems. UNIX is considered as the first OS to use a full-screen editor and editing code online by various people becomes easy in UNIX.

- **Other features:**

  Some other features of UNIX include:-

  - UNIX is free
  - The file system is hierarchical by which accessing and retrieving files become easy
  - The performance of UNIX is better than Windows NT
  - Stable database access
  - Better handling of internet and intranet in servers
  - Internet-client and file server are better managed using Java in UNIX

1.5 DISADVANTAGES OF UNIX OPERATING SYSTEM

1.5.1 Not user friendly:

Novice user has difficulty in using UNIX. Most of the work in UNIX is done by using commands in CLI so beginner has to remember a different type of commands. UNIX is solely made for programmers

and not for beginner users. Some experience people also feel difficulty in using commands because some commands are very different from their name.

1.5.2 Poor documentation:

There is not any proper documentation available for UNIX. If the user gets any problem then he has to consult some expert and getting online help from the documentation is very difficult. If you compare this with Windows and macOS then you will get proper and easy to follow the documentation that is available online.

1.5.3 Cryptic commands:

Most of the commands in UNIX use cryptic words. It is difficult for the casual user to get the idea of working of command. Some commands use special characters and understanding commands for newbie programmers become difficult. If you use any wrong character in the command then your system will start doing unknown works that may also delete or change some data from your computer. Some commands in UNIX work in a combination of other commands so if you forget any command then your work cannot be completed.

1.5.4 High learning curve:

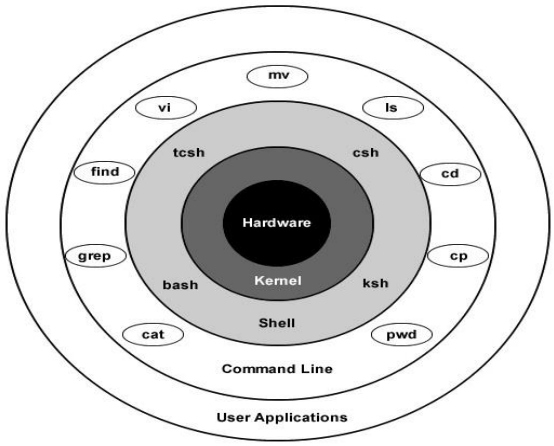
The commands in UNIX is much more difficult than commands in Windows MS-DOS and mac OS. It also becomes difficult to remember the commands. You have to go through the documentation every time you have to use any complex command. The new UNIX OS comes with GUI but still, most of the work is done using CLI. The apps and software in UNIX are also not very popular and you have to learn them before using this software.

- **Other drawbacks:**

  Some other drawbacks of UNIX include:-

  - Some drivers are not available for the software
  - The UI of UNIX is not so much attractive
  - Some variants of UNIX requires most memory usage
  - As UNIX has fewer users so most game companies not make games for UNIX.
  - UNIX has very fewer games available than Windows and Mac OS

1.6 ARCHITECTURE OF UNIX SYSTEM



The Architecture of the UNIX system is divided into 4 major components. They are:

- ✓ The Kernel
- ✓ The Shell
- ✓ Files and Processes
- ✓ System Calls
- ✓ **The Kernel**

  The Kernel is the heart of the Operating System.

  It interface between Shell and Hardware.

  It performs Low Level Task.

  Eg: Device Management, Memory Management etc.

- ✓ **The Shell**

  The Shell is a collection of UNIX Commands.

  The Shell acts as an interface between the user and the kernel.

  The Shell is a Command Line Interpreter(CLI)–Translates the commands provided by the user and converts it into a language that is understood by the Kernel.

  Only one Kernel running on the system, but several shells in action-one for each user who is logged in.

  Eg: C Shell, Bourne Shell, Korn Shell etc.

- ✓ **Files and Processes**

  A File is an array of bytes and can contain anything.

  All the data in UNIX is organized into files.

  A Process is a program file under execution.

  Files and Processes belong to a separate hierarchical structure.

- ✓ **System Calls**

  UNIX is written in C. Thousands of commands in the system uses a handful of functions called System Calls to communicate with the kernel.

1.7 KERNEL AND SHELL

Both the Shell and the Kernel are the Parts of this Operating System. These Both Parts are used for performing any Operation on the System. When a user gives his Command for Performing Any Operation, then the Request Will goes to the Shell Parts, The Shell Parts is also called as the Interpreter which translates the Human Program into the Machine Language and then the Request will be transferred to the Kernel. So that Shell is just called as the interpreter of the Commands this converts the Request of the User into the Machine Language. Kernel is also called as the heart of the Operating System and every operation are performed by using the Kernel, When the Kernel Receives the Request from the Shell then this will Process the Request and Display the Results on the Screen.

1.7.1 The Various Operations performed by the Kernel :

- It Controls the State the Process Means it checks whether the Process is running or Process is waiting for the Request of the user.
- Provides the Memory for the Processes those are Running on the System Means Kernel Runs the Allocation and De-allocation Process, First When we Request for the service then the Kernel will Provides the Memory to the Process and after that he also Release the Memory which is Given to a Process.
- The Kernel also Maintains a Time table for all the Processes those are Running Means the Kernel also Prepare the Schedule Time means this will provide the Time to various Process of the CPU and the Kernel also Puts the Waiting and Suspended Jobs into the different Memory Area
- When a Kernel determines that the Logical Memory doesn‘t fit to Store the Programs. Then he uses the Concept of the Physical Memory which Will Stores the Programs into Temporary Manner. Means the Physical Memory of the System can be used as Temporary Memory.
- Kernel also maintains all the files those are Stored into the Computer System and the Kernel Also Stores all the Files into the System as no one can read or Write the Files without any Permissions. So that the Kernel System also Provides us the Facility to use the Passwords and also all the Files are Stored into the Particular Manner.

1.7.2 Shell:

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

Every unix system has atleast one shell.A shell is a program that sites on kernel and acts as an agent or interface between the users and the kernel and hence the hardware.A shell is a command interpreter or processor at which the user can type in any unix command.

1.7.3 Types of shells:

There are different types of shells available. Some of them are.

(a) **The bourne shell(sh):**

This is the first mojar shell to be developed and is named after its author, Stephen bourne.

- This is the most common shell and is disturbuted as the standard shell on almost all unix systems.
- This is the most common shell on almost all unix systems.

(b) **C shell(csh):**

- Bill joy, developed this shell at VCB as a part of the BSD release .
- It is called c shell because is syntax and usuage is very similar to the ‘c’ programming language.

(c) **Korn shell(ksh):**

- This shell was developed by david korn at AT &T bell labs.

- It has the both features of bourne shell and c shell and is one of the widely used shells.

(d) Bourne-again shell(bash):

- This shell was developed by BFOX and C raney at free software foundation.
- This is a free ware shell

1.7.4 The shell prompt:

- Successful login into a unix system is indicated by the appearance of a prompt called the shell prompt or system prompt on the terminal.
- The character that appears as a prompt depends on the shell used.
- **$(dollar)-> bourne and korn shells(sh,bash,ksh)**
- **%(percent)-> C shell (sh and tesh)**
- **#(hash)->any shell as root.**

1.8 UNIX COMMANDS

Unix has large number of commands. Types of Unix commands are two types .

1. External commands
2. Internal commands

External commands: A command with an independent existence in the form of a separate file is called an external command. Ex: cat and ls . these are independently existed in a directory called the / bin directory . When external commands and given the shell reaches these command files with the help of 'PATH' variable.

Internal commands: A command that does not have an independent existence is called an internal command. Ex. Echo,mkdir,cd,etc; The routines for internal commands will be a part of another program (or) routine. Ex. Echo command is internal command will be a part of another program(or) routine is the part of shell's routine "sh".

1.8.1 Some basic commands:- Unix has several hundreds of commands with it. some of them are.

1. echo
2. tput
3. lty
4. who
5. uname
6. date
7. cal
8. calendar
9. passwd
10. lock
11. banner
12. cat etc..;

The echo command:-

Use:-

- ➢ The echo command is used to display messages.
- ➢ It takes zero,one(or) more no of arguments.
- ➢ Arguments may be given either as a series of individual symbols or as a string with in a pair of double quotes(“ “).
- ➢ Example:

  1 . $echo #Blank line is displayed

  $

  2. $echo Iam studying information technology

  $ I am studying information technology

  3. $echo “Iam studying unix programming”.

  Iam studying unix programming.

The tput command:-

Use:-

- ➢ This command is used to control movement of the cursor on the screen as well as to add the certain features like blinking ,boldface and underlining to the displayed message on the screen.
- ➢ Examples :-
  - $tput clear
- ➢ Clears the screen and puts the cursor at the left top of the screen
  - $tput cup 10 20
- ➢ This command along with cup a argument and certain co-ordinates values is used to position the cursor at any required position on the screen.
  - In the above example the cursor will be placed at the tenth and twentieth column on the screen.
  - $tput lines

  48

  $
- ➢ To know the current no of rows on terminal.
  - $tput cols

  142

  $

The tty command:-

In unix, every terminal is associated with a special file, called the device file. All the device files will be present in /dev directory.

Use:- Examples:

- $tty /dev/tty 01

$

➢ Here tty01 is the file name and will be available in /dev directory.

The who command:-

Use:-

The user can know the login details of the current users by using the who command .

Ex:- $who

Root console nov 19 09:35

Mgv tty01 nov 19 09:40

dvm tty02 nov 19 09:45

➢ The first column shows the name of the users,the second column shows devicename and third column shows login time.

Ex:- $who am i

Mgv tty01 nov 19 9:40

➢ The self-login details of a user can be obtained in the above example

The uname command:-

Use: -

➢ using this command one can know the details of one's unix system.

➢ Unix has many variants, when this command is used .it gives the name of Unix system being used by the user.

Ex:- $uname

Linux

$

The date command:-

➢ Using this command, the user can display the current date along with the time nearest to the second.

Ex:-

• $date Sat jan 18 11:58:00 lst 2018

$

• $date +%m

09

$

➢ Displays only month in numeric form.

• %date +%h

Sep

$

➢ Displays the name of the month

The cal command:- Use:-

➢ This command is used to print the calendar of a specific month or a specific year.

➢ When this command is used without any arguments the calendar of the current month of the current year will be printed.

Ex:-

$cal 02 2018

Jan 2018

Su	mo	tue	wed	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	31	

$

The passwd command:-

- ➢ As unix is a multi-user system due to which there is always a security threat.
- ➢ The simplest and most widely used by all individual users is use of passwds.
- ➢ For every new user,the administrator ,permita or authorities them by assigning unique password to each.

Use:-

- ➢ A user can change his/her password by using this command.

Ex:- $passwd: *****

Old passwd: *******

New passwd: *******

$

When a user attempts to change his/her password the system first asks for old password ,if it is correctly entered the system asks twice to enter new password and srts it as current password

The lock command:-

Use:-

- ➢ The lock command is used for locking a session for any required amount of time.

- ➢ This command is used generally without any arguments.

- ➢ By default the user can lock it for 30 min.
- ➢ The locking period can be changed by assigning a different value forthe system variable DEFLOGOUT.
- ➢ When the lock command is given, the terminal asks for a password twice.

Eg: $lock -45 #locks for 45 min

The banner command:

- ➢ It is used to display banners or pasters.
- ➢ This command simply produces a blowup version of the characters that are supplied with the command.
- ➢ Ex: $ banner hello

The cat command:

- ➢ This is used to create small unix files. A file can be created by writing a command line.
- ➢ Eg: $ cat > review

  A > symbol following the command

  Means that the output

  Goes to the file name following it

  <ctrl -d>
- ➢ In the above example, 'review' is the file name after executing $cat > review, the $ disappears and the presents '>' symbol which means the system is ready to accept the contents to be placed in the file.
- ➢ If the file contents are over to exit, we have to use the command <ctrl -d>

The bc command:

- ➢ The bc command is both a calculator and a small language for writingnumerical programs.
- ➢ By using this one can perform all the usual arithmetic operations as well as change of bases in the range of 2 -16
- ➢ Ex: $ bc

Sqrt 55

7 Quit

$

The spell and ispell command:

- ➢ The spell command is the first program that was developed to check for words that are wrongly spelt in a document.

- ➢ This command displays a list of misspelled words in the document used as argument.
- ➢ Eg: $cat spell – ux

This is example

I am using cat command

*

$ spell spell

. ux Example

Command

$

Ispell:

The ispell command is an interactive spell – check program available in linux . when used, this command displays a screen full of information in tree sections.

Eg: $ispell spell.ux

This is example I am using cat command

1. Example
2. Exempler
3. Exampled

I. Ignore ignore all

II. Replace replace all

III. Add exit

The man command:

The man command prints the details of any command of a utility on the screen.

Ex: $man pwd

Pwd(1) user commands pwd(1)

Name

Pwd - print working directory pwd(1)

Synopsis

Pwd

Description

Pwd prints the pathname of the working directory Notes

Command substitution:

In unix, it is possible to run a command within another command.

Ex: The date command can be run within the echo command by writing command as

$ echo today the date is ‘date’

Today the date is the june 7 16:25:00 2024

when more trend is executed the shell while parsing the parameters list of the echo command treats the words that are back quoted as a command executes it and substitutes the result of this execution at the corresponding position in the parameter list. This process is known as command substitution.

Multiple commands:

- ➢ Normally a single command is given to the shell at its prompt. However there are many situations when more than one command is given in a single command line.
- ➢ One of the ways of giving the multiple commands is to use a semicolon (;) between successive commands.
- ➢ Ex: echo "giving multiple command" ; who ; who, date does not mutually interact with each other in any manner.
- ➢ They are executed one after the other, from left to right as they appear in the command line.

1.9 SUMMARY

In this unit, we have learn about the basic concept of unix operating system and its Basic Features, advantages, Basic Architecture of Unix system, Kernel, Types of shells: Bourne shell, C Shell, korn shell, and basic unix commands

1.10 CHECK YOUR PROGRESS

- ➢ **What do you mean by Unix OS?**

  ……
  ……
  ……
  ……
  ……

- ✓ **What are the Advantages and Disadvantages of UNIX operating system?**

  ……
  ……
  ……
  ……
  ……

- ✓ **Discuss the major components of the UNIX system Architecture**

  ……
  ……
  ……
  ……
  ……

- ✓ **Discuss the role of kernel in operating system**

  ……
  ……
  ……

……
……

- ✓ **List the Features of UNIX operating system**

……
……
……
……
……

- ✓ **What is a System call in UNIX?**

……
……
……
……
……

Further Reading:

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education

UNIT-2 WORKING WITH FILES AND DIRECTORIES

Structure

2.1 INTRODUCTION

File system is one of the important pillars of UNIX system. UNIX treats everything (even a user, program, directory etc) as a file. Hence, organizing and managing the file system is very important topic to study. Here, we will discuss how to create directories, moving around the file system, listing filenames, various file attributes, security related issues like file permission, changing the file permission etc

2.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- How to work with **Files and Directories** of UNIX operating system
- Concept of Unix file **and Directories** system
- Know about creating the files,
- Listing Files and Directories,
- file permissions and directory permissions,
- How to removing a file ,
- Learn about Mkdir, ls, pwd and cd,echo and cat,wc, ls commands
- tput command,

2.2 THE FILE

The file is a container for storing information. Unlike old DOS files, a UNIX file does not contain eof (end-of-file) character. It contains only the information stored by the user. All the attributes of a file are kept in a separate area of the hard disk, which can be accessible by only the kernel. UNIX treats directories and devices also as files. Even the physical devices like hard disk, memory, CD-ROM, printer, modem etc. are treated as files.

In UNIX, files are divided into three categories –

- Ordinary file
- Directory file
- Device file

2.3 UNIX FILES SYSTEM

The UNIX file system is a methodology for logically organizing and storing large quantities of data such that the system is easy to manage. A file can be informally defined as a collection of data, which can be logically viewed as a stream of bytes (i.e. characters).

A file is the smallest unit of storage in the UNIX file system. By contrast, a file system consists of files, relationships to other files, as well as the attributes of each file. File attributes are information

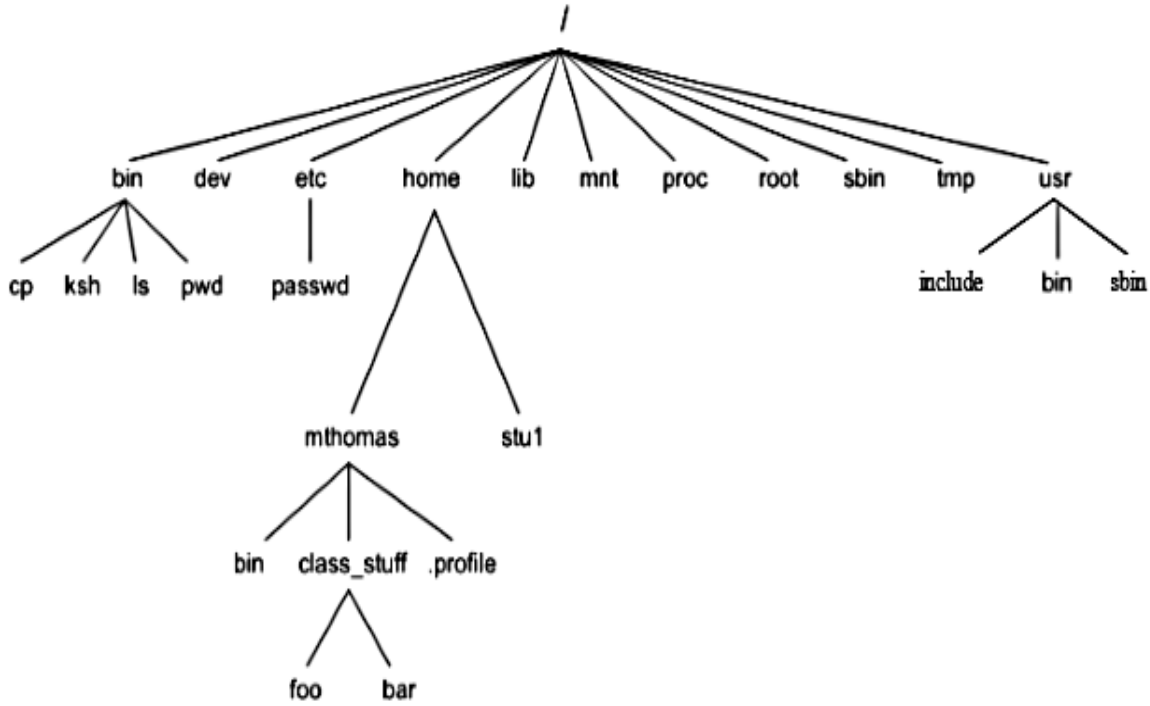
relating to the file, but do not include the data contained within a file. File attributes for a generic operating system might include (but are not limited to):

- a file type (i.e. what kind of data is in the file)
- a file name (which may or may not include an extension)
- a physical file size
- a file owner
- file protection/privacy capability
- file time stamp (time and date created/modified)

Additionally, file systems provide tools which allow the manipulation of files, provide a logical organization as well as provide services which map the logical organization of files to physical devices.

From the beginner‘s perspective, the UNIX file system is essentially composed of files and directories. Directories are special files that may contain other files.

The UNIX file system has a hierarchical (or tree-like) structure with its highest-level directory called root (denoted by /, pronounced slash). Immediately below the root level directory are several subdirectories, most of which contain system files. Below this can exist system files, application files, and/or user data files. Similar to the concept of the process parent child relationship, all files on a UNIX system are related to one another. That is, files also have a parent-child existence. Thus, all files (except one) share a common parental link, the top-most file (i.e. /) being the exception. Below is a diagram (slice) of a "typical" UNIX file system. As you can see, the top-most directory is / (slashes), with the directories directly beneath being system directories. Note that as UNIX implementations and vendors vary, so will this file system hierarchy. However, the organization of most file systems is similar.



While this diagram is not all inclusive, the following system files (i.e. directories) are present in most UNIX file systems:

- bin - short for binaries, this is the directory where many commonly used executable commands reside
- dev - contains device specific files
- etc - contains system configuration files
- home - contains user directories and files
- lib - contains all library files
- mnt - contains device files related to mounted devices
- proc - contains files related to system processes
- root - the root users' home directory (note this is different than /)
- sbin - system binary files reside here. If there is no sbin directory on your system, these files most likely reside in etc
- tmp - storage for temporary files which are periodically removed from the filesystem
- usr - also contains executable commands

2.4 FILES CREATING COMMANDS

We can create a file in the Linux/Unix system using the terminal. In the Linux/Unix system, there are the following ways available to creating files.

1. Using the touch command
2. Using the cat command
3. Using redirection operator
4. Using the echo command
5. Using the heredoc
6. Using the dd command

2.4.1 Touch command: The touch command is used to create file/files without any content and update the access date or modification date of a file or directory in the Linux system. This is the simplest way to create a file in Linux/Unix using the terminal.

Syntax: touch [OPTION]…[FILE]

Example: Create empty files called _file1‘ and _file2‘

$ touch file1 file2

Option	Description
-a	Change the access time of a file
-c, –no-create	Check file is available or not, if not available then prevent creating a file

Option	Description
-f	ignored
-m	Change the modification time of a file
-t STAMP	Use specified time instead of the current time
–help	Display help and exit
–version	Display the version information and exit.

2.4.2 The cat command :

The cat (concatenate) command is used to create, view, concatenate files in the Linux operating system. The touch command is also used to create a file in a Linux system without content whereas the cat creates files with some content. The cat command reads the content of a file and prompts it.

Syntax: cat [OPTION]…[FILE]

Option	Description
-A, –show-all	Show all content of a file
-b, –number-nonblank	Display number of non-empty lines overrides -n
-n, –number	Display number of all output lines
-T, –show-tabs	Display this help and exit
–help	Display this help and exit
–version	Display version information and exit

To create a file with some content, we use the cat command and file name after that write some content and press CTRL + C

Example :

$ cat > file.txt

Display contents of the files using the cat command in the Linux/Unix system.

The cat command is also used to view the contents of the file.

Redirection operator.

In the Linux/Unix system a redirection operator is also used to create a file.

Example :

$ > file.txt

2.4.3 Echo Command :

The echo command is also used to create a new file in the Linux system.

Create a new file without contents in the Linux/unix system using the echo command.

To create a file without contents, we use the echo command with a redirection operator followed by the file name as shown below.

Example :

$ echo > file.txt

2.3.4 heredoc command :

heredoc stands for here document. The heredoc delimiter is a type of redirection. It allows passing multiple lines of input to a command.

The general syntax of heredoc. Important

Command << Heredoc_delimiter

multiple lines contents...

heredoc_delimiter

Create a file with multiple lines of contents using a heredoc delimiter in the Linux/unix system.

To create a file using heredoc, we use the cat command with heredoc delimiter in the Linux system as shown below.

Example :

$ cat << heredoc_delimiter < file_name

2.3.4. dd command :

The dd command is mainly used to converts and copy files. To check more details about the dd command. We can also create a large file using the dd command.Create a large file in the Linux/unix system using the dd command. To create a large file, we use the dd command as shown below.

Example :

$ dd if = /dev/zero of = file.test bs =1 count =0 seek = 2G

2.5 LISTING THE FILES

There are many ways to list files and display information about them on Linux systems. Most of the commands list files within individual directories while others reach as deeply into a file system as you care to look.The primary command for listing files is, of course, **ls**. This command, however, has an extensive number of options for finding and listing the files

Syntax of `ls` command in UNIX

ls [option] [file/directory]

Example: **ls**
Result: Lists the names of files in your default directory, in alphabetical order.

Example: **ls -l**
Result: Gives a "long listing" of the files in your directory. In addition to the file name, the long listing shows protection information, file owner, number of characters in file, and the date and time of the last change to the file.

Example: **ls -a**
Result: Causes all your files to be listed, including those files that begin with a period (i.e., hidden files).

For more information, type **man ls** at the Unix system prompt.

Options	Description
-l	known as a long format that displays detailed information about files and directories.
-a	Represent all files Include hidden files and directories in the listing.
-t	Sort files and directories by their last modification time, displaying the most recently modified ones first.
-r	known as reverse order which is used to reverse the default order of listing.
-S	Sort files and directories by their sizes, listing the largest ones first.
-R	List files and directories recursively, including subdirectories.
-i	known as inode which displays the index number (inode) of each file and directory.

Options	Description
-g	known as group which displays the group ownership of files and directories instead of the owner.
-h	Print file sizes in human-readable format (e.g., 1K, 234M, 2G).
-d	List directories themselves, rather than their contents.

ls –l option:

Listing File Attributes The –l option of ls command is used for listing the various attributes like permissions, size, ownership etc. of a file. The output of ls –l is referred to as the listing. The –l option can be combined with other options for displaying other attributes, or ordering the list in a different sequence. The command ls use inode of a file to fetch its attributes. Consider the following example of ls –l which displays seven attributes of all files in the current directory

$ ls -l

total 144

-rw-rw-r-- 1 john john 280 Jan 30 09:56 caseEx.sh

-rw-rw-r-- 1 john john 104 Feb 3 06:40 cmdArg.sh

-rw-rw-r-- 1 john john 199 Jan 29 10:58 ifEx.sh

-rw-rw-r-- 1 john john 217 Jan 19 09:25 logfile

drwxrwxr-x 2 john john 4096 Feb 6 05:48 myDir

-rwxrwxr-x 1 john john 29 Jan 22 10:04 myFirstShell

-rw-rw-r-- 1 john john 43 Jan 22 10:44 second.sh

The output of ls –l starts with total 144, indicates that a total of 144 blocks are occupied by these files on disk. The 7 types of attributes/fields displayed by the command are discussed below –

- File Type and Permissions: The first column shows the type and permissions associated with each file. If the first character in this column is – (hypen), then it is an ordinary file. On the other hand, if the first character is d, then it is a directory. Then, there is a series of r, w, x and – (hyphen) indicating file permissions read, write and execute. The hyphen indicates absence of particular permission.
- Links: The second column indicates the number of links associated with the file. It is a number of filenames maintained by the system of that file. Usually for ordinary files, it will be just 1. But for directories, it will be number of files contained within that directory (including current directory).

- Ownership: The creator of the file would be its owner. In the third column, it shows john as the owner. The owner has full authority to modify the contents and permissions of a file. Similarly, the owner of a directory can create modify or remove files in that directory.
- Group Ownership: While opening a user account, a system administrator assigns the user to some group. The fourth column represents the group owner of that file.
- File Size: Size of the file in bytes is shown as fifth column. The size is only a character count of the file, but not the amount of space it occupies in the disk.
- Last Modification Time: The 6th, 7th and 8th columns indicate the last modification time of the file. A file is said to be modified only if its contents have changed. If you change only the permission or ownership of the file, its last modification time field will not get affected.
- Filename: The last field is the name of the file, usually in ASCII collating sequence.

2.6 FILE OWNERSHIP

The person who creates a file will be the owner of that file. The login name of that person will be showed as the owner when ls –l option is used. The group to which the person belongs to, will be the group owner of the file. If you copy someone's file, you will be the owner only for that copy. One cannot create files in other's home directory, because one may not have permission to do so. Several users may belong to a single group. The files created by group members will have the same group owner. However, the privileges of the group are set by the owner of the file, but not by group members. When the system administrator creates a user account, he has to assign the following parameters to the user:

- The user-id (UID) – both its name and numeric representation
- The group-id (GID) – both its name and numeric representation

The file /etc/passwd maintains the UID (both name and number) and GID (only the number). The file /etc/group contains the GID (both name and number). To know your UID and GID, the id command can be used as –

$id

uid=821(chetana) gid=822(STAFF

2.7 FILE PERMISSIONS

UNIX has a simple and well-defined system of assigning permissions to files. UNIX follows three-tiered file protection system to determine a file's access rights. Consider the ls –l command to view the permissions of few files:

$ ls –l chap02 dept.lst shell1.sh

-rwxr-xr-- 1 john richard 20500 Sep 29 11:53 chap02

-rwxr-xr-x 1 john richard 850 Oct 02 10:12 dept.lst

-rw-rw-rw- 1 john richard 48 Nov 07 08:03 shell1.sh

Observe the first column representing permission for the file unit 02. Here, the first character says whether the file is ordinary file or directory. So, leaving it apart, consider next 9 characters as a group of 3 characters each

– rwx r-x r--

Each group represents a category viz. owner, group owner and others (or world) respectively. Every group contains any of the characters r, w, x and -. The meaning of these is as below –

- r: indicates read permission – means, cat command can display the file
- w: indicates write permission – file can be edited with an editor
- x: indicates execute permission – the file can be executed as a program
- -: indicates absence of the corresponding permission

Generally, the owner of a file will have all the three permissions. In the above example, the group owner of the file unit 02 has only read and executes permissions. The public or others or world has only read permission

2.8 CHANGING FILE PERMISSIONS (chmod):

A file or directory is created with a default set of permissions. Generally, in the default setting, write permission is not given to group and others. That is, only the user (owner) the file can write a file. But, read permission will be given to all. The **chmod** (change mode) command is used for assigning/removing different permissions to/from category (user, group, others). This command can be run only by the user (owner) and the super-user (admin). The **chmod** command can be used in two ways –

- In a relative manner by specifying the changes to the current permissions
- In an absolute manner by specifying the final permissions

Consider the permissions of an existing file test as below –

$ls –l test

-rw-r--r-- 1 john richard 853 Sep 5 23:38 test

It is observed here that, by default the execute permission is not there even for the user (owner).

2.9 RELATIVE PERMISSIONS

When changing permissions in a relative manner, chmod changes only the permissions specified in the command line and leaves the other permissions unchanged. The syntax is–

chmod category operation permission filenames

The argument for chmod is an expression consisting of some letters and symbols describing user category and type of permission being assigned/removed. The expression contains three components:

- User category (user: u, group: g, others: o, All: a)
- The operation to be performed (assign: +, remove: –, assign absolute permission: =)
- The type of permission (read: r, write: w, execute: x)

Now, consider the example of the file test taken before, and assign execute permission to it as below–

$ chmod u+x test #assign(+) x(execute) to u(user)

$ ls –l test

-rwxr--r-- 1 john richard 853 Sep 5 23:38 test

Now, the user john got permission to execute the file test. If you want to assign execute permission on file test to group and others also, then use the command as –

$ chmod ugo+x test #assign(+) x to u(user, group, others)

$ ls –l test

-rwxr-xr-x 1 john richard 853 Sep 5 23:38 test

The string ugo can be replaced by a indicating all as shown below –

$ chmod a+x test #assign(+) x to a(all)

$ ls –l test

-rwxr-xr-x 1 john Richard 853 Sep 5 23:38 test

When you are willing to assign a particular permission to all, then even the character a can be omitted as below –

$ chmod +x test #assign(+) x to all

When same set of permissions has to be assigned to more than one file, then we can give multiple files separated with space as –

$ chmod u+x test test1 test2

To remove a permission, the – (hyphen or minus) operator is used. For example, to remove read permission from group and other, we can do as below –

$ ls –l test #check current status

-rwxr-xr-x 1 john richard 853 Sep 5 23:38 test

$ chmod go-r test #remove r permission from group & others

$ ls –l test #verify

-rwx—x--x 1 john richard 853 Sep 5 23:38 test

Multiple expressions separated by comma can be given to chmod command. For example, to remove the execute permission from all and then to assign read permission to group and others, a single statement can be used as –

$ chmod a-x, go+r test

$ ls –l test -rw-r--r-- 1 john richard 853 Sep 5 23:38 test

More than one permission can also be set as below –

$chmod o+wx test

$ ls –l test -rw-r--rwx 1 john richard 853 Sep 5 23:38 test

2.10 ABSOLUTE PERMISSIONS

Irrespective of existing permissions for a file, we may need to assign a new set of permissions. That is, we wish to set all nine permission bits explicitly. This is known as absolute permissions. For this purpose, chmod uses a string of three octal numbers. Various permissions are given a specific digit as below –

- Read permission – 4

- Write permission – 2
- Execute permission – 1

Every possible combination of three different permissions is shown in binary representation in Table .

Table 2.1 Digits used for absolute pathnames

Binary	Octal	Permission	Significance
000	0	- - -	No permission
001	1	- - x	Execute only
010	2	- w -	Write only
011	3	- wx	Write and execute
100	4	r- -	Read only
101	5	r – x	Read and execute
110	6	rw-	Read and write
111	7	rwx	Read, write and execute

Using chmod Recursively (–R)

The chmod command can be used to apply required permissions on all files (and files within subdirectory) in a given directory. It is done using –R option as below –

$chmod –R a+x ShellPgms

This makes all files and subdirectories found in the tree-walk (starting from ShellPgms directory, includes all files in subdirectories) executable by all users. One can provide multiple directory and filenames for this purpose. If chmod has to be applied on home directory tree, one can use any one of the following –

$chmod –R 755 . #works on hidden files also

$chmod –R a+x * #leaves out hidden files

CHANGING FILE OWNERSHIP

It has been discussed that the creator of a file will be the owner. And the group to which user belongs to, will be the group owner. But, except the owner, other group members cannot alter the permissions of any file. UNIX provides two commands for changing the ownership of a file or directory viz. chown and chgrp. The usage of these commands differs from system to system. On BSD – based systems, only the system administrator can change the ownership of a file using chown. On the other systems, only the owner can use chown and chgrp.

Berkeley Software Distribution (BSD) was a Unix operating system derivative developed and distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, from 1977 to 1995.)

2.12 CHANGING FILE OWNER (chown)

The syntax of chown command in BSD-based systems is –

chown options owner [:group] files

To use chown command in BSD-based systems, we need the super-user permission. For that, the su command is used as below –

$su Password: ******** (this is root password)

#_ (This is another shell)

The command su lets us to acquire superuser status (Note that # is the prompt for admin). Now, try to change the ownership of a file note which is currently owned by john as below–

ls –l note -rwxr----x 1 john metal 347 May 10 20:30 note

chown ricky note # ls –l note

-rwxr----x 1 ricky metal 347 May 10 20:30 note

Here, the ownership of the file note has been changed from john to ricky. The file permissions previously assigned to john will now be of ricky. Now onwards, john is not the owner of this file and he cannot read/write this file.

2.13. CHANGING GROUP OWNER (chgrp)

By default, the group owner of a file is the group to which the owner belongs to. But, the chgrp command changes a file's group owner. A user can be a member of more than one group. So, in BSD – based systems, group can be changed only among the set of groups for which the user is a member.

Assume that john is a member of two groups metal and dba. And he has created a file dept.txt in metal group. He can change the group owner as below –

$ls –l dept.txt

-rw-r--r-- 1 john metal 129 Jun 8 16:42 dept.txt

$chgrp dba dept.txt

-rw-r--r-- 1 john dba 129 Jun 8 16:42 dept.txt

When the user is not a member of particular group, he cannot change the group owner of any file to that group. Only superuser can do so.

2.14 SECURITY AND FILE PERMISSIONS

There are certain security issues while using chmod command. Consider removing all the permissions for a file as below –

$chmod 000 test

$ls –l test

---------- 1 john richard 830 May 10 20.30 test

Now the file is virtually useless, as no one can do anything with it. But, user can still delete this file. At the same time, one must be aware that giving all permissions to everyone is dangerous. That is, having the following statement makes the file readable/writable/executable for everyone.

$chmod 777 test

So, anyone can modify the contents of the file and it is a threat on security.

- MORE FILE ATTRIBUTES

  Every file is associated with a table that contains various attributes of a it, except its name and contents. This table is called as inode (index node) and accessed by the inode number. The inode contains following attributes of a file –

- File type (ordinary, directory, device etc)
- File permissions
- Number of links
- The UID of the owner
- The GID of the group owner
- File size in bytes
- Date and time of last modification
- Date and time of last access
- Date and time of last change of the inode
- An array of pointer that keep track of all disk blocks used by the file

2.15 DEFAULT FILE AND DIRECTORY PERMISSIONS (umask)

When you create files and directories, the permissions assigned to them depend on the default setting of the system. The UNIX system has the following default permissions for all files and directories–

- rw-rw-rw- (octal 666) for ordinary files
- rwxrwxrwx (octal 777) for directories

This default setting is transformed by subtracting the user mask from it to remove one or more permissions. To understand this, we should know the current value of mask by using umask command without arguments as –

$umask

0022

This is an octal number which has to be subtracted from the system default to obtain the actual default. Hence, the actual default of files will be –

- 666 – 022 = 644 for ordinary files
- 777 – 022 = 755 for directories

Hence, when we create a new file on this system, the default permission of it would be – rw-r--r--

The umask is a shell built-in command, though it exists as an external command. A user can use this command to set a new default. For example,

$umask 000

The above command sets the umask to 000 and hence, any new file created will have the permissions as 666-000 = 666 permission. That is, it would be rw-rw-rw-, which is dangerous because anyone can write the file.

Similarly, if you set

$umask 666

Then, all files created will have permission as 666-666 =000, and it will be a useless file. So, the mask has to be set carefully.

2.16 LOCATING FILES (find):

The find command is one of the powerful tools of the UNIX system. It recursively examines a directory tree to search for files matching some criteria, and then takes some action on the selected files. The syntax of this command is –

find path_list selection_criteria action

The working of find command is as below –

- Initially, it recursively examines all files in the directories specified in path_list
- Then it matches each file for one or more selection_criteria
- Finally, it takes some action on those selected files.

2.17 DISPLAYING & CONCATENATING (COMBINING) FILES (MORE)

Enables examination of a continuous text one screenful at a time on a terminal. It normally pauses after each screenful, printing -- More -- at the bottom of the screen. Press RETURN to display one more line. Press the SPACE BAR to display another screenful. Press the letter Q to stop displaying the file.

Example: **more newfile**

Result: Displays the contents of "newfile" one screen ("page") at a time.

For more information about this command, type **man more** at the Unix system prompt.

cat-- Displays the contents of a file on your terminal.

Example: **cat newfile**

Result: Displays the contents of the file "newfile" on your terminal.

Example: **cat newfile oldfile**

Result: Displays the contents of two files–"newfile" and "oldfile"–on your terminal as one continuous display.

While a file is being displayed, you can interrupt the output by pressing **CTRL + C** and return to the Unix system prompt. **CTRL + S** suspends the terminal display of the file and the processing of the command. To resume display, press **CTRL + Q**. The interrupted command displays lines beginning at the point at which processing was interrupted.

The cat command is also used to concatenate (combine) files and put them into another file. If you concatenate files to another one that already exists, the existing contents are permanently lost.

Example: **cat fileone filetwo filethree > newfile**

Result: Links together three files–fileone, filetwo, and filethree–into a new file called "newfile." The original files remain intact.

For more information about the cat command, type **man cat** at the Unix system prompt.

2.18 COPYING FILES(CP)

cp–Makes copies of your files. You can use it to make copies of files in your default directory, to copy files from one directory to another directory, or to copy files from other devices.

Example: **cp fileone file two**

Result: Copies the contents of fileone to a file named filetwo. Two separate files now exist.

Example: **cp /usr/neighbor/testfile .**

Result: Copies the file testfile from the directory /user/neighbor to your Unix account. The period(.) at the end of the command line indicates that the file is to be copied to your current working directory and the name will remain the same.

To copy a file from another user's directory on Unix, you must know the person's username.

Example: **cp ~username/file1 yourfile**

Result: Copies the file "file1" from user to your Unix account. The name of the file in your directory becomes yourfile. (Protections must be set for file to be readable by you in the other user's directory in order to be able to copy the file.)

For more information, type **man cp** at the Unix system prompt.

2.19 DELETING/REMOVING A FILES

rm stands for **remove** here. rm command is used to remove objects such as files, directories, symbolic links and so on from the file system like UNIX. To be more precise, rm removes references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names). **By default, it does not remove directories.** This command normally works silently and you should be very careful while running **rm** command because once you delete the files then you are not able to recover the contents of files and directories.

Syntax:

rm [OPTION]... FILE...

Let us consider 5 files having name **a.txt, b.txt** and so on till **e.txt**.

$ ls

a.txt b.txt c.txt d.txt e.txt

Removing one file at a time

$ rm a.txt

$ ls

b.txt c.txt d.txt e.txt

Removing more than one file at a time

$ rm b.txt c.txt

$ ls

d.txt e.txt

Note: No output is produced by **rm**, since it typically only generates messages in the case of an error. **Options: 1. -i (Interactive Deletion):** Like in cp, the -i option makes the command ask the user for confirmation before removing each file, you have to press **y** for confirm deletion, any other key leaves the file un-deleted.

$ rm -i d.txt

rm: remove regular empty file 'd.txt'? y

$ ls

e.txt

2. -f (Force Deletion): rm prompts for confirmation removal if a file is **write protected**. The **-f** option overrides this minor protection and removes the file forcefully.

$ ls -l

total 0
-r--r--r--+ 1 User User 0 Jan 2 22:56 e.txt

$ rm e.txt

rm: remove write-protected regular empty file 'e.txt'? n

$ ls

e.txt

$ rm -f e.txt

$ ls

Note: -f option of rm command will not work for write-protect directories. **3. -r (Recursive Deletion):** With **-r(or -R)** option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory. At each stage it deletes everything it finds. Normally, **rm** wouldn't delete the directories but when used with this option, it will delete. Below is the tree of directories and files:

$ ls

A

$ cd A

$ ls

B C

$ ls B

a.txt b.txt

$ ls C

c.txt d.txt

Now, deletion from **A** directory(as parent directory) will be done as:

$ rm *
rm: cannot remove 'B': Is a directory
rm: cannot remove 'C': Is a directory

$ rm -r *

$ ls

Every directory and file inside **A** directory is deleted. **4. –version:** This option is used to display the version of **rm** which is currently running on your system.

Applications of wc Command Delete file whose name starting with a hyphen symbol (-): To remove a file whose name begins with a dash ("-"), you can specify a double dash ("–") separately before the file name. This extra dash is necessary so that rm does not misinterpret the file name as an option. Let say there is a file name **-file.txt**, to delete this file write command as:

$ ls

-file.txt

$ rm -file.txt

rm: unknown option – l

Try 'rm ./-file.txt' to remove the file '-file.txt'.

Try 'rm --help' for more information.

$ rm -- -file.txt

$ ls

2.20 RENAMING FILES

mv–This command changes the identification (name) of one or more files.

Example: **mv oldfile newfile**
Result: Changes the name of the file "oldfile" to "newfile." Only one file will exist.

Example: **mv oldfile bin/newfile**
Result: Changes the name of the file "oldfile" to "newfile" and places it in the directory /bin. Only one file will exist.

For more information, type **man mv** at the Unix system prompt.

Printing from Unix

The **lpr** command prints files on Unix. Use the -Pqueuename option to select a printer.

Example: **lpr -Ppittprint sample.file**
Result: This is the default output. Single-sided output, one page-worth of text per side, portrait format. Output is queued to the Pitt Print Stations.

The Unix operating system is case sensitive; type all commands in lower-case letters unless noted otherwise.

WHAT IS IN A FILE NAME?

In most of the UNIX systems now a days, a filename can consists of 255 characters. They can have any printable and non-printable characters except / and the NULL character. But, as UNIX uses special characters like $, ?, *, &, ` etc for different purpose, it is better to avoid certain characters. So, ideally, a filename can contain alphabets, digits and special characters dot (.), hyphen (-) and underscore (_).

UNIX does not impose any rule for framing the extensions for filenames. Even shell scripts do not require .sh as extension. It is used only for the convention. But, underlying programming languages

like C requires extension. Hence, in UNIX, a filename can contain any number of dots – say, a.b.c.d.e is a valid filename in UNIX. A filename can begin/end with a dot. But, UNIX is case sensitive and same is maintained in naming the files as well. Thus, test, Test, TEST all are different files.

2.21 DIRECTORIES

A directory contains no data, but it keeps some information about the files and subdirectories that it contains. The UNIX file system is organized with a number of directories and subdirectories. A user also can create them, as and when required. Usually, a group of related files are kept in a single directory. Sometimes, files with same name are kept in different directories.A directory file contains an entry for every file and subdirectory it has. Each such entry has two components .**The filename is** a unique identification number for the file or directory (called as the inode number) **,**Thus, a directory actually do not contain the file itself, rather, it contains only the file name and a number. One cannot write into a directory file. But, the actions like creating a file, removing a file etc. makes kernel to update the corresponding directory by creating/removing filename and inode number associated with that file.

2.22 DEVICE FILE

The activities like printing files, installing softwares from CD-ROM, taking backup of files into a tape/drive etc. are performed by reading or writing the file representing the device. For example, when you are printing a file in a printer, you are writing a file associated with printer.

Device filenames are generally found inside a single directory structure, /dev. A device file is not a stream of characters. In fact, it does not contain anything. The operation of a device is completely managed by the attributes of its associated file. The kernel identifies a device from its attributes and then uses them to operate the device

2.23 THE PARENT–CHILD RELATIONSHIP

All files in UNIX are related to each other. The file system in UNIX is a collection of all types (ordinary, directory and device files) related files organized in a hierarchical structure. A UNIX file system has root represented by /, which serves as reference point for all files. The root is actually a directory and it is different from the user-id root used by the system admin to log-in.

The root directory (/) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and files under them. Figure 2.1 shows an example of UNIX file system tree structure. Here, bin, dev etc. are directories under root (/). And, mthomas, stu1 are subdirectories under home

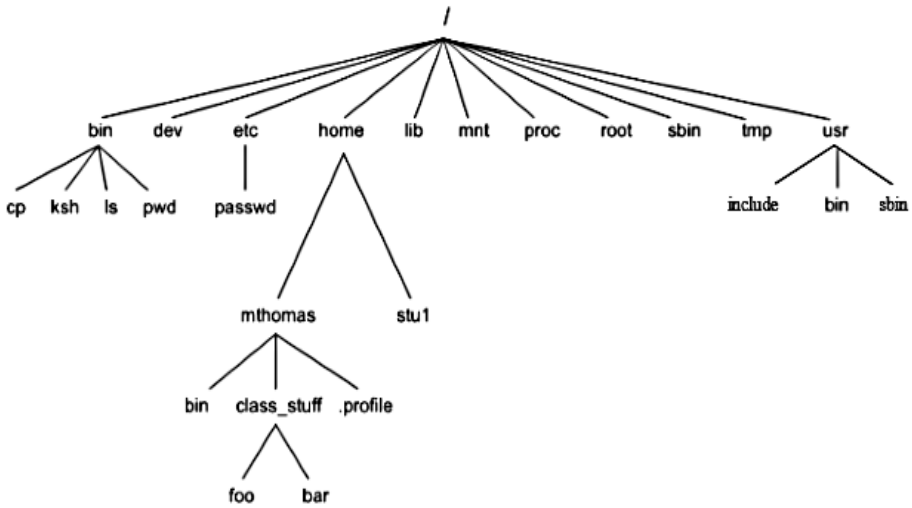


Figure The UNIX File System Tree

Every file, apart from root, must have a parent, and there will be a parent-child relationship path from root to file. In the Figure 2.1, cp is child of bin and bin is child of /. That is, / is grandparent of cp. Note that, in a parent-child relationship, parent is always a directory.

2.24 THE HOME DIRECTORY

When a user logs into the system, UNIX places him into a directory called as home directory. It is created by the system when a user account is opened. If you have logged in with a user name john, then, your home directory would be /home/john. This can be viewed using the shell variable $HOME as – $echo $HOME /home/john # the first / represents root directory

The path displayed here is known as absolute pathname, which is a sequence of all directory names separated by slash (/) starting from root. A file foo located in a home directory of the user can be referred as $HOME/foo. In some of the shells, it can be referred as ~/foo. Here, the ~ symbol can be used to refer any other's file also. For example, if there is a file called foo in another user richard's directory also, then it can be referred as ~richard/foo.

Note that, a ~ (tilde) followed by / refers to one's own home directory, but when followed by a string (like ~richard), it refers to home directory represented by that string.

Include

2.25 CHECKING YOUR CURRENT DIRECTORY (pwd):

Once a user logs in to the UNIX system, it places him in a specific directory (usually home directory) of the file system. Though a user can move from one directory to other, for a given moment of time, he will be in one directory, known as current directory. To know current directory, the pwd (print working directory) command is used. The pwd command displays the absolute pathname as below – $pwd /home/john

One can navigate the file system using cd command

2.26 CHANGING THE CURRENT DIRECTORY (cd):

The cd command is used to move around the file system by changing the directory. This command can be used in three different ways –

- If the user Shyam is in his home directory and would like to move to subdirectory called as progs, then the command should be given as –

  $ cd progs # user is moved to progs directory now

  $ pwd # verify this using pwd

  /home/shyam/progs

- If the user would like to shift some other directory (but not his own subdirectory), then absolute path name can be given.

  For example,

  $ pwd #check current directory

  /home/shyam/progs $ cd /bin # change directory to /bin

  $ pwd # verify new location

  /bin

- When cd is used without argument, it will take the user back to his home directory, that is where he had logged into.

  Ex1.

  $ pwd #check current directory

  /home/shyam/progs

  $ cd #no arguments given

  $ pwd /home/shyam #back to home directory

- cd command may fail when you do not have permission to access a directory

2.27 MAKING DIRECTORIES (mkdir):

A directory can be created using the command mkdir. Various ways of using this command is listed here. The name of the directory to be created has to be given as argument.

For example,

$mkdir docs

One can create more than one directory with a single mkdir command as below –

$mkdir docs progs db # three directories created

- A directory tree can be created as –

  $mkdir test test/prgms test/data

Initially the directory test will be created. Then two subdirectories prgms and data will be created under test. Note that, while creating such directory tree, the parent directory name has to be given first.

- Sometimes, trying to create a directory may fail and error message may get displayed as –

  $mkdir myDir

  mkdir: Failed to make directory “myDir”; Permission denied

The reason may be any of these –

- The directory myDir may already exist
- There may be any ordinary file by that name in the current directory
- The permission is set for current directory so as to not to create new files/directories within it.

2.28 rmdir: REMOVING DIRECTORIES

To remove (or delete) a directory, the rmdir command is used. Few important points about this command are discussed here –

- A directory has to be empty before removing it. That is, it should not contain any files or subdirectories.
- To remove one directory, use statement like –

$rmdir test #removes the directory test

More than one directory (even in tree structure) can be removed at a time.

Note that, here *prgms* and data are two subdirectories under test. They should be given first and then the parent directory test has to be mentioned.

To remove a particular directory, you should be hierarchically above that directory. That is you cannot remove any directory that is above your current directory and also, you cannot remove your current directory as well.The commands mkdir and rmdir can work only in the directories owned by the user. But they cannot be implemented on the directories of some other user

2.29 TPUT COMMAND

tput command is used to query the terminfo terminal database and check if that terminal supports a specific feature.tput command accepts the terminal commands and output the control code sequences for that terminal. Using tput you can control the color and cursor of your terminal .

1. Set the Cursor Position using tput cup

   You can move the cursor to a specific row and column using tput cup. Following example positions the cursor at row 2 and column 3.

```
$ tput cup 2 3
```

2. Clear the Screen Using tput clear

   If you are in middle of the terminal screen, tput clear will clear the screen and put you at the top of the terminal screen.

```
$ tput clear
```

3. Get the Number of Columns and Lines of a Terminal

   To display the number of columns of your terminal screen, do the following.

```
$ tput cols
```

   Following displays number of lines of your terminal screen.

```
$ tput lines
```

4. Execute Multiple tput Commands

   tput allows you to run set of commands in a single time. For example, if you want to clear the screen and set cursor to a particular position, do the following.

```
$ tput -S <<END
> clear
> cup 2 4
> END
```

5. Change the Terminal Background Color using tput setb

   Using tput, the background color of the screen can be changed as shown below.

```
$ tput setb 4
```

Note: You can also change only your prompt color using Bash custom prompt PS1.

6. Change the Foreground Color using tput setf

You can also change the foreground color of the terminal as shown below.

```
$ tput setf 4
```

Note: If you set foreground and background to the same color you cannot see the cursor. So, to reset, execute "tput reset".

7. Turn On and Turn Off Highlighting

tput allows you to turn on and turn off the text high lighting. When you turn it ton, new text in the terminal gets bold.

```
$ tput bold
```

When you turn it off, new text in the terminal returns to normal display.

```
$ tput sgr0
```

In the below example, it bolds the particular text 'guide' by turning on and turning off highlighting accordingly.

```
$ echo `tput bold`guide`tput sgr0`
guide
```

8. Underline Text using smul and rmul

Start the underline mode:

```
$ tput smul
```

Stop the underline mode:

```
$ tput rmul
```

In the below example, it underlines the 'guide' text by using smul and rmul capabilities of tput.

```
$ echo `tput smul`guide`tput rmul`
guide
```

9. Hide and Unhide the Cursor using civis and cnorm

Hide the cursor:

```
$ tput civis
```

Display the cursor:

```
$ tput cnorm
```

2.30 SUMMARY

In this unit, we have learn about how to work with file and directories, concept of Unix file system, creating files, Listing Files and Directories, masking file permissions, directory permissions, removing a file forcibly, Mkdir, ls, pwd and cd,echo and cat,wc,ls –l, other useful variations of ls, tput command, etc.

2.31 CHECK YOUR PROGRESS

- **What do you mean by file system of Unix OS?**

  ……
  ……
  ……

- **What are the file creating command in UNIX operating system?**

  ……
  ……
  ……
  ……
  ……

- **Discuss how listing the file**

  ……
  ……
  ……
  ……
  ……

- **What do you mean by file permission in unix operating system**

  ……
  ……
  ……
  ……
  ……

- **List the directory command of UNIX operating system**

  ……
  ……
  ……
  ……
  ……

- **What is tput command in UNIX?**

  ……
  ……
  ……
  ……
  ……

1) What is a file system? Explain Unix file system with neat diagram
2) What is a file ? Explain its attributes and permissions with examples
3) Write short notes on Inodes.
4) Explain briefly about file permissions with examples
5) Mention and Explain the commands used for changing the owner of a file as well as the group of a file.
6) Illustrate file directories with neat diagram and explain Directory handling commands supported by Unix. (pwd, mkdir, cd, rmdir)
7) Explain the basics of file, directories and file names ?
8) Give a short note on file and mention various types of file and explain File handling commands (cat, rm, cp, mv, wc)
9) Briefly explain how to change file permissions, Changing the owner of a file ,Changing the group of a file.
10) Briefly explain about ls command with all options and examples. 11)Explain about the file command-knowing the file type.

Further Reading:

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education

Unit-3 UNIX FILE SYSTEM INTRODUCTION TO UNIX FILE SYSTEM

Structure

3.11 Check your progress

3.1 INTRODUCTION

Computers can store information on various storage media, such as Magnetic disks, Magnetic tapes, and Optical disks. A file is a named collection of related information that is recorded on secondary storage. File is a memory block of secondary storage device like disk, magnetic tape that logically stores related records permanently. Operating system provides various components to manage the data on a disk system permanently. Such as..

File manager: It manages the allocation of disk-space for the storage of user files.

Buffer manager: It is responsible for fetching data from disk storage into main memory buffers for processing, and then writing the updated data back onto the disk.

Disk Space manager: It is responsible for managing disk space on disk.

3.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- Concept of Unix file system
- Know about **Types of File Blocks on a disk**,
- Learn about **File Access Methods**,
- Learn about Disk related commands
- tput command,

Unix File system-Boot block, super block, Inode table, data blocks, How Unix access files, storage files, Disk related commands: checking disk free space, dfsapce, du, ulimit.

3.2 FILE SYSTEM

Provides a powerful & flexible way to organize and manage user information. File system is a Group of files and relevant information are organized in a well defined order and stored on a Hard Disk . Operating System defines a File System on Devices which is usually Hierarchical file system including UNIX.

Directory- It is a File that keeps a list of other files and sub directories on the File System . UNIX has single File system. Devices are mounted into this File System. The disk space allotted to a linux/unix file system is made up of “blocks” , each of which are 512 bytes

To find out block size on your file system, use the **cmchk** command.

$cmchk

BSIZE=

2048

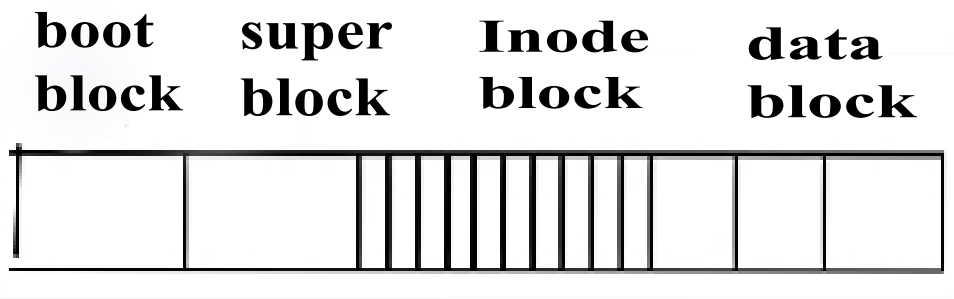


Fig: Disk File Block Structure

3.3 TYPES OF FILE BLOCKS ON A DISK

All blocks belonging to the file system are logically divided into :

1) Boot block
2) Super block
3) Inode block
4) Data block

Boot block: It is the first block in the storage disk, numbered 0 , is called the boot block. It is normally unused by the file system and is set aside for booting procedure. It represents the beginning of the file system. It contains a program called "Boost Strap Loader". This program is executed during 'booting' process to load the file system into the main memory. (Booting - means starting process of system when computer is turned ON.

Super block: It is the second block of every file system and is numbered 1. It describes the entire file system and its state. It is used to control the allocation of blocks . It contents of super block are:

a) The size of file and status of file name
b) Details of free blocks and their addresses. A file occupies a minimum of 1 Block, even it has one character in it. A block size can be of 512 bytes or multiple of 512 bytes.
c) Size of inode section of the file system.
d) It specifies "how large the file system is, how many maximum files it can accommodate, how many more files it can accommodate, how many more files can create etc."

Inode block: It contains most information regarding the files. Every file will have an entry in this area, identified by a 64-bit structure referred to as I-node. The information related to all the files (not the contents) is stored in an inode table on the disk. For each file, there is an inode entry in the table. Each entry is made up of 64 bytes in the table. These details are..

1) Owner of the file
2) Group to which the owner belongs
3) Type of a file
4) File access permission
5) Date and time of last access
6) Date and time of last modifications

7) No of links to the file

8) Size of the file

9) Address of blocks where the file is physically present.

Data blocks: Contains the actual file contents. An allocated block can belong to only one file in the file system.This block can't be used for storing any other file's contents unless the file which it originally belonged is deleted.

3.4 THE BASICS OF A FILE

FILE : File is a sequence of Records or Bytes . A byte is a small chunk of information, typically 8 - bits long. Here, a byte is equivalent to a character. Data is usually stored in the form of records.

RECORD: It is a collection of Fields. Each record consists of a collection of related data values or items, where each value is formed of one or more bytes.

Files are the building blocks of any operating system. Every file has some properties.

TYPES OF FILES: A file in a UNIX system having the following types:

- Regular file / Ordinary file (-)
- Directory file (d)
- Device or Special file (c or b)
- Hidden files or Dot files (.)
- FIFO file (f)
- Link file (l)
- Socket file (s)

1. **Ordinary files** – An ordinary file is a file on the system that contains data, text, or program instructions.
   - Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
   - Always located within/under a directory file.
   - Do not contain other files.
2. Directories – Directories store both special and ordinary files. UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components.
   i) The Filename
   ii) A unique identification number for the file or directory (called the inode number)

Features:

- Branching points in the hierarchical tree.
- Used to organize groups of files.
- May contain ordinary files, special files or other directories.

- Never contain “real” information which you would work with (such as text). Basically, just used for organizing files.
- All files are descendants of the root directory, (named /) located at the top of the tree.
- In long-format output of ls –l , this type of file is specified by the “d” symbol.

Device or Special Files – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Ouput (I/O) operations. On UNIX systems there are two types of special files for each device, character special files and block special files.

- **Character special file:** A file related to I/O and used to model serial I/O devices like Terminals, Printers & Networks.
- **Block Special file**: A file related to memory and used to model devices like Disk drives and Magnetic tapes.

Hidden files or Dot files : A dot(.) character also used to construct a file system. Any file name beginning with a .(dot) character is called a hidden file a dot file. These are used to store some specific information regarding configuration of system. Ex: .profile, .bashrc etc.

FIFO file : used for making communication between two or more application programs.

3.5 FILE ACCESS

File access methods in an operating system are the techniques and processes used to read from and write to files stored on a computer’s storage devices. There are several ways to access this information in the file. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

These methods determine how data is organized, retrieved, and modified within a file system. Understanding file access methods is crucial for efficient data management and system performance. In this article, we are going to discuss different types of methods to access the file.

3.6 FILE ACCESS METHODS

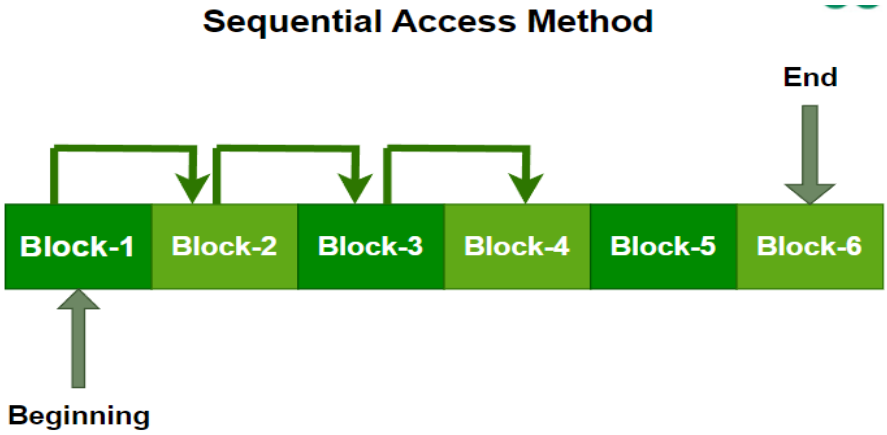
There are three ways to access a file in a computer system:

- Sequential-Access
- Direct Access
- Index sequential Method

Sequential Access

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, the editor and compiler usually access the file in this fashion.

Read and write make up the bulk of the operation on a file. A read operation -*read next*- reads the next position of the file and automatically advances a file pointer, which keeps track of the I/O location. Similarly, for the -write *next*- append to the end of the file and advance to the newly written material.



Key Points related to Sequential Access

- Data is accessed from one record right after another record in an order.
- When we use the read command, it moves ahead pointer by one.
- When we use the write command, it will allocate memory and move the pointer to the end of the file.
- Such a method is reasonable for tape.

Advantages of Sequential Access Method

- It is simple to implement this file access mechanism.
- It uses lexicographic order to quickly access the next entry.
- It is suitable for applications that require access to all records in a file, in a specific order.
- It is less prone to data corruption as the data is written sequentially and not randomly.
- It is a more efficient method for reading large files, as it only reads the required data and does not waste time reading unnecessary data.
- It is a reliable method for backup and restore operations, as the data is stored sequentially and can be easily restored if required.

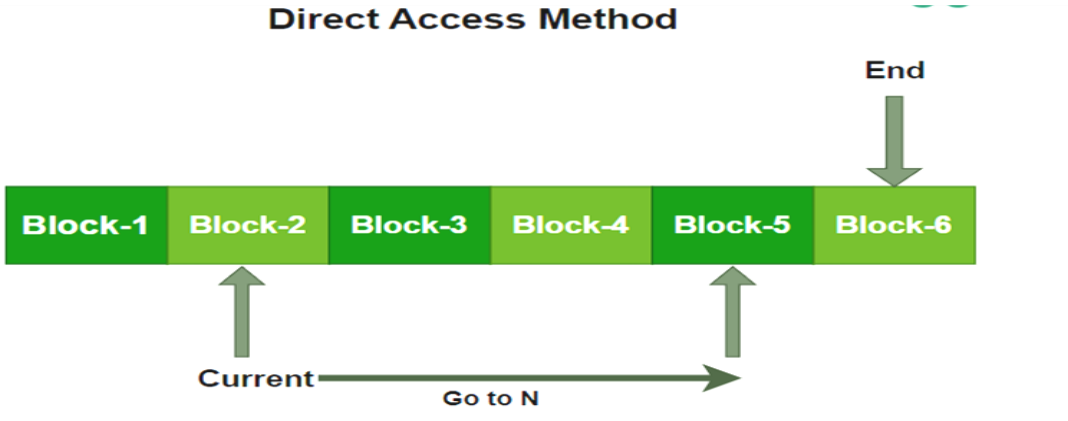
Disadvantages of Sequential Access Method

- If the file record that needs to be accessed next is not present next to the current record, this type of file access method is slow.
- Moving a sizable chunk of the file may be necessary to insert a new record.
- It does not allow for quick access to specific records in the file. The entire file must be searched sequentially to find a specific record, which can be time-consuming.
- It is not well-suited for applications that require frequent updates or modifications to the file. Updating or inserting a record in the middle of a large file can be a slow and cumbersome process.
- Sequential access can also result in wasted storage space if records are of varying lengths. The space between records cannot be used by other records, which can result in inefficient use of storage.

Direct Access Method

Another method is *direct access method* also known as *relative access method*. A fixed-length logical record that allows the program to read and write record rapidly. in no particular order. The direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59, and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file.

A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on.



Direct Access Method

Advantages of Direct Access Method

- The files can be immediately accessed decreasing the average access time.
- In the direct access method, in order to access a block, there is no need of traversing all the blocks present before it.

Disadvantages of Direct Access Method

- **Complex Implementation**: Implementing direct access can be complex, requiring sophisticated algorithms and data structures to manage and locate records efficiently.
- **Higher Storage Overhead**: Direct access methods often require additional storage for maintaining data location information (such as pointers or address tables), which can increase the overall storage requirements.

Index Sequential method

It is the other method of accessing a file that is built on the top of the sequential access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index, and then by the help of pointer we access the file directly.

Key Points Related to Index Sequential Method

- It is built on top of Sequential access.
- It control the pointer by using index.

Advantages of Index Sequential Method

- **Efficient Searching**: Index sequential method allows for quick searches through the index.
- **Balanced Performance**: It combines the simplicity of sequential access with the speed of direct access, offering a balanced approach that can handle various types of data access needs efficiently.
- **Flexibility**: This method allows both sequential and random access to data, making it versatile for different types of applications, such as batch processing and real-time querying.
- **Improved Data Management**: Indexing helps in better organization and management of data. It makes data retrieval faster and more efficient, especially in large databases.
- **Reduced Access Time**: By using an index to directly locate data blocks, the time spent searching for data within large datasets is significantly reduced.

Disadvantages of Index Sequential Method

- **Complex Implementation**: The index sequential method is more complex to implement and maintain compared to simple sequential access methods.
- **Additional Storage**: Indexes require additional storage space, which can be significant for large datasets. This extra space can sometimes offset the benefits of faster access.
- **Update Overhead**: Updating the data can be more time-consuming because both the data and the indexes need to be updated. This can lead to increased processing time for insertions, deletions, and modifications.
- **Index Maintenance**: Keeping the index up to date requires regular maintenance, especially in dynamic environments where data changes frequently. This can add to the system's overhead.

Other Way of File Access

1. **Relative Record Access**

   Relative record access is a file access method used in operating systems where records are accessed relative to the current position of the file pointer. In this method, records are located based on their position relative to the current record, rather than by a specific address or key value.

   Key Points Related to Relative Record Access

   - Relative record access is a random access method that allows records to be accessed based on their position relative to the current record.
   - This method is efficient for accessing individual records but may not be suitable for files that require frequent updates or random access to specific records.
   - Relative record access requires fixed-length records and may not be flexible enough for some applications.
   - This method is useful for processing records in a specific order or for files that are accessed sequentially.

Advantages of Relative Record Access

- **Random Access:** Relative record access allows random access to records in a file. The system can access any record at a specific offset from the current position of the file pointer.
- **Efficient Retrieval:** Since the system only needs to read the current record and any records that need to be skipped, relative record access is more efficient than sequential access for accessing individual records.
- **Useful for Sequential Processing:** Relative record access is useful for processing records in a specific order. For example, if the records are sorted in a specific order, the system can access the next or previous record relative to the current position of the file pointer.

Disadvantages of Relative Record Access

- **Fixed Record Length:** Relative record access requires fixed-length records. If the records are of varying length, it may be necessary to use padding to ensure that each record is the same length.
- **Limited Flexibility:** Relative record access is not very flexible. It is difficult to insert or delete records in the middle of a file without disrupting the relative positions of other records.
- **Limited Application:** Relative record access is best suited for files that are accessed sequentially or with some regularity, but it may not be appropriate for files that are frequently updated or require random access to specific records.

2. Content Addressable Access

Content-addressable access (CAA) is a file access method used in operating systems that allows records or blocks to be accessed based on their content rather than their address. In this method, a hash function is used to calculate a unique key for each record or block, and the system can access any record or block by specifying its key.

Keys in Content-Addressable Access

- **Unique:** Each record or block has a unique key that is generated using a hash function.
- **Calculated based on content:** The key is calculated based on the content of the record or block, rather than its location or address.

Key Points Related to Content-Addressable Access

- Content-addressable access is a file access method that allows records or blocks to be accessed based on their content rather than their address.
- CAA uses a hash function to generate a unique key for each record or block.
- CAA is efficient for searching large databases or file systems and is more flexible than other access methods.
- CAA requires additional overhead for calculating the hash function and may have collisions or limited key space.

Advantages of Content-Addressable Access

- **Efficient Search:** CAA is ideal for searching large databases or file systems because it allows for efficient searching based on the content of the records or blocks.

- **Flexibility:** CAA is more flexible than other access methods because it allows for easy insertion and deletion of records or blocks.
- **Data Integrity**: CAA ensures data integrity because each record or block has a unique key that is generated based on its content.

Disadvantages of Content-Addressable Access

- **Overhead:** CAA requires additional overhead because the hash function must be calculated for each record or block.
- **Collision:** There is a possibility of collision where two records or blocks can have the same key. This can be minimized by using a good hash function, but it cannot be completely eliminated.
- **Limited Key Space:** The key space is limited by the size of the hash function used, which can lead to collisions and other issues.

3.7 STORAGE FILES

In file storage, data is stored in files. The files are organized in folders, and the folders are organized under a hierarchy of directories and subdirectories. To locate a file, all you or your computer system need is the path, from directory to subdirectory to folder to file.

Hierarchical file storage works well with easily organized amounts of structured data. But, as the number of files grows, the file retrieval process can become cumbersome and time-consuming. Scaling requires adding more hardware devices or continually replacing these with higher-capacity devices, both of which can get expensive.

To some extent, you can mitigate these scaling and performance issues with cloud-based file storage services. These services allow multiple users to access and share the same file data located in off-site data centers (the cloud). You simply pay a monthly subscription fee to store your file data in the cloud, and you can easily scale-up capacity and specify your data performance and protection criteria.

Moreover, you eliminate the expense of maintaining your own onsite hardware since this infrastructure is managed and maintained by the cloud service provider (CSP) in its data center. This is also known as Infrastructure-as-a-Service (IaaS).

File storage has been a storage technique for decades, it's familiar to virtually every computer user, and it's well-suited to storing and organizing transactional data or manageable structured data volumes that can be neatly stored in a database in a disk drive on a server.

However, many organizations are now struggling to manage mounting volumes of web-based digital content or unstructured data. If you need to store large or unstructured data volumes, you should consider block-based or object-based storage that organizes and accesses data differently.

Depending on the various speed and performance requirements of your IT operations and various applications, you might require a combination of these approaches.

File storage, also called file-level or file-based storage, is a hierarchical storage methodology used to organize and store data on a computer hard drive or on network-attached storage (NAS) device.

Block storage

Block storage offers greater storage efficiency, more efficient use of available storage hardware and faster performance than file storage. Block storage breaks a file into equally sized chunks or blocks of data and stores each block separately under a unique address.

Rather than conforming to a rigid directory, subdirectory or folder structure, blocks can be stored anywhere in the system. To access any file, the server's operating system uses the unique address to pull the blocks back together into the file, which takes less time than navigating through directories and file hierarchies to access a file.

Block storage works well for critical business applications, transactional databases and virtual machines that require low-latency and minimal delay. It also gives you more granular access to data and consistent performance.

Object storage

Object-based storage has emerged as a preferred method for data archiving and backing-up today's digital communications, unstructured media and web content like email, videos, image files, web pages and sensor data produced by the Internet of Things (IoT). It's also ideal for archiving data that does not change frequently like static files, such as large volumes of pharmaceutical data or music, image and video files.

Objects are discrete units of data that is stored in a structurally flat data environment. Again, there are no folders, directories or complex hierarchies; instead, each object is a simple, self-contained repository that includes the data, metadata (descriptive information associated with an object), and a unique identifying ID number. This information enables an application to locate and access the object.

You can aggregate object storage devices into larger storage pools and distribute these storage pools across locations. This allows for unlimited scale and improved data resiliency and disaster recovery. Objects can be stored locally, but most often reside on cloud servers, with accessibility from anywhere in the world.

File storage benefits

If your organization requires a centralized, easily accessible and affordable way to store files and folders, file-level storage is a good approach. The benefits of file storage include the following:

Simplicity: File storage is the simplest, most familiar and most straightforward approach to organizing files and folders on a computer's hard drive or NAS device. You simply name files, tag them with metadata, and store them in folders under a hierarchy of directories and subdirectories. It is not necessary to write applications or code to access your data.

File sharing: File storage is ideal for centralizing and sharing files on a local area network (LAN). Files stored on a NAS device are easily accessible by any computer on the network that has the appropriate permission rights.

Common protocols: File storage uses common file-level protocols such as server message block (SMB), common internet file system (CIFS), or network file system (NFS). If you use a Windows or Linux® operating system (or both), standard protocols like SMB/CIFS and NFS will allow you to read and write files to a Windows-based or Linux-based server over your LAN.

Data protection: Storing files on a separate, LAN-connected storage device offers you a level of data protection should your network computer experience a failure. Cloud-based file storage services provide more data protection and disaster recovery by replicating data files across multiple, geographically dispersed data centers.

Affordability: File storage that uses a NAS device allows you to move files off of expensive

computing hardware and onto a more affordable LAN-connected storage device. Moreover, if you choose to subscribe to a cloud file-storage service, you eliminate the expense of onsite hardware upgrades and the associated ongoing maintenance and operation costs.

File storage use cases

File storage is a good solution for a wide variety of data needs, including the following:

Local file sharing: If your data storage needs are consistent and straightforward, such as storing and sharing files with team members in the office, consider the simplicity of file-level storage.

Centralized file collaboration: If you upload, store and share files in a centralized library, located onsite, off-site or in the cloud, you can easily collaborate on files with internal and external users or with invited guests outside of your network.

Archiving or storage: You can cost-effectively archive files on NAS devices in a small data center environment or subscribe to a cloud-based file storage service to store and archive your data.

Backup or disaster recovery: You can store backups securely on separate, LAN-connected storage devices. Or you can subscribe to a cloud-based file storage service to replicate your data files across multiple, geographically dispersed data centers and gain the additional data protection of distance and redundancy.

Cloud-based file storage or file storage hosting

Today's communications are rapidly moving to the cloud to gain the benefits of a shared storage approach that inherently optimizes scale and costs. You can reduce your organization's onsite IT infrastructure by using low-cost cloud storage while keeping your data accessible when you need it.

Similar to an onsite file storage system, cloud-based file storage, also called file storage hosting, allows multiple users to share the same file data. But instead of storing data files locally on a NAS device, you can store these files off-site in data centers like the cloud and access them via the internet.

With cloud-based file storage, you no longer need to refresh your storage hardware every 3 to 5 years or to budget for the installation, maintenance and personnel you need to manage it. Instead, you simply subscribe to a cloud storage service for a predictable monthly or annual fee. You can reduce your IT personnel or reassign these technical resources to more revenue-producing areas of your business.

Storing file data in the cloud also enables you to scale up capacity as needed and on demand. Cloud-based file storage services typically offer simple, predefined tiers with varying levels of storage capacity and workload performance requirements (total number of input or output operations per second, or IOPS), and data protection and replication to other data centers for business continuity, all for a predictable monthly fee. Or you can increase or decrease IOPS and expand data volumes dynamically, paying only for what you use.

There are strategic benefits to subscription-based cloud storage services, particularly for multi-site and larger organizations. These include ease of sharing across a network of locations, disaster recovery and the ease of adding innovations and technologies that emerge in the future.

3.8 DISK RELATED COMMANDS (du, df):

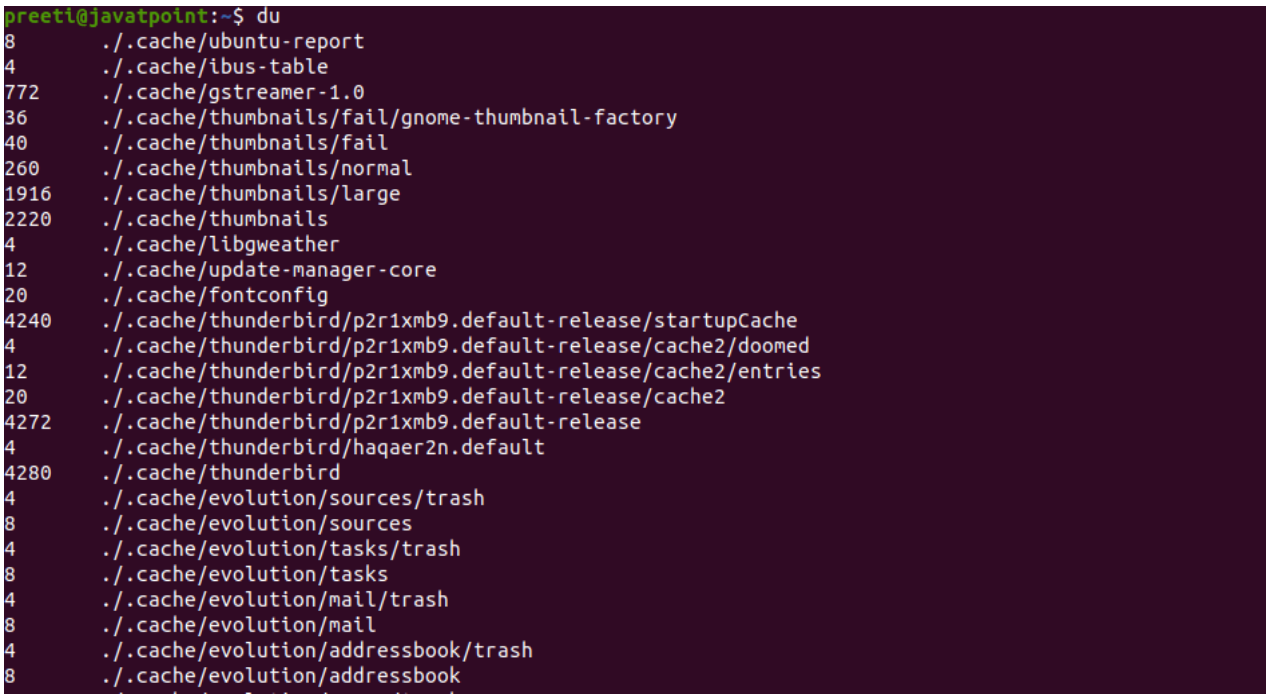
There are many ways by which we can check the Linux system disk space. We can use a third-party app which displays the available disk space. Another way is the command-line by the Linux Terminal, some of which are df and du, where **du** means disk space used and **df** means disk space free.

1. Using du Command

   We can check the disk space with the help of the **du** command. The full-form of **du** is "Disk Usage." The du command shows the usage of disk. The du command will show you the disk usage for specific directories in Linux.

   Syntax:

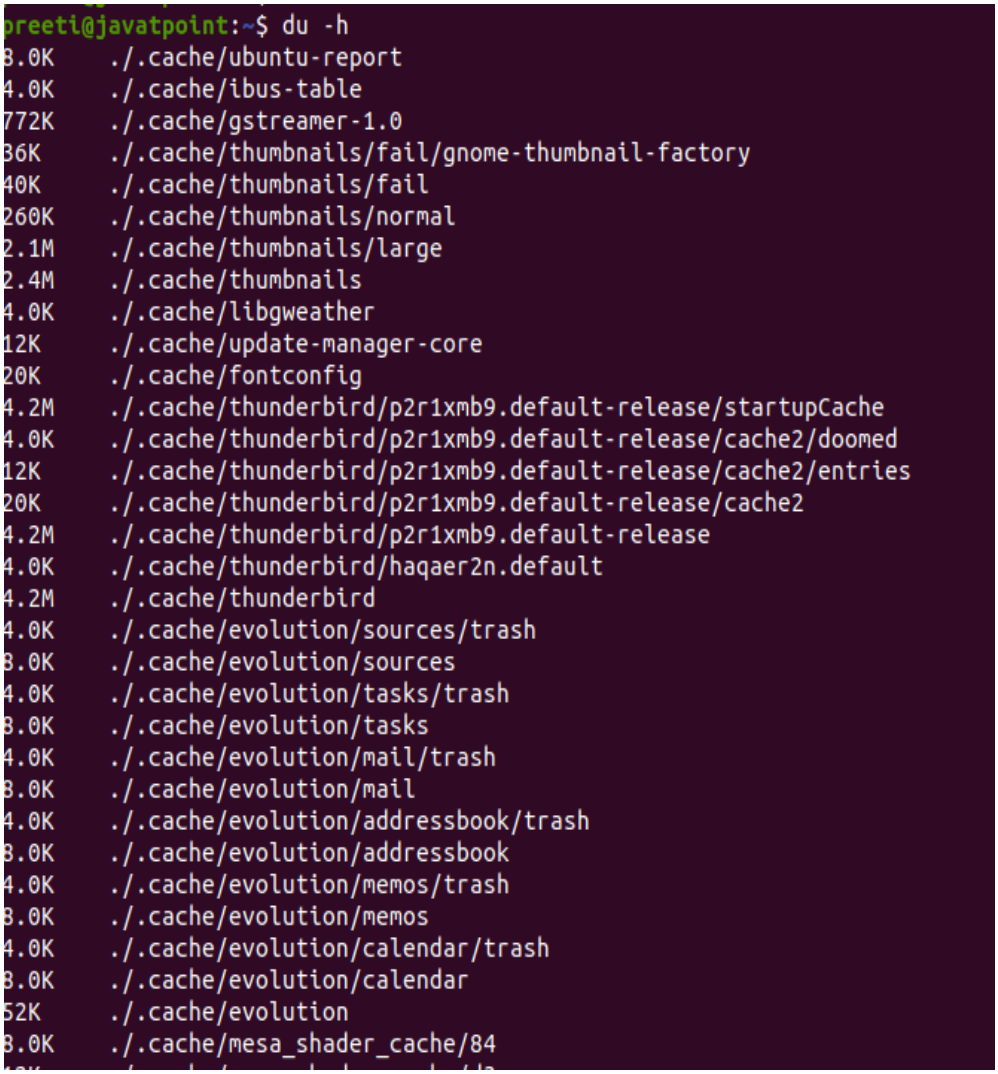
   $ du



   du -h: - This will show you the disk usage in a human-readable format for each directory and subdirectory.

   Syntax:

   $ du -h



- **du -a: -** This will show you the disk usage for all the files.

  Syntax:

  $ du -a

```
6       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/1888DBE9FDE3F5E43509061D28642BE92574637D
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/F542AD421AD560AFBD19150CCEAB14D48E20CC29
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/AA11AA495162816094A789282730CD89B5CEF360
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/D3BAF85CF1A0A003786BC368FA4C01A0CFC69760
6       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/27020560BF2E32C3DC617D13F76E16079B6D2328
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/DE89BB2C409C552E350E9A4430A6143E985DDEC3
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/4C05B6AE88057B362A53302560EDA457F4D26AD6
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/C665046DE7421543785DA0470FA61E370E33B160
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/475530181FF861972443644A59749CEF598334E0
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/49FEA3FC1754F4F29C4ABB266742B700460291C5
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/ADCCFA9B4A4C883B870957E617D8A18ACA017E56
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/27DD5B5357A46E0F8C7B2AF012D35974C12A5463
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/FCB047459109CB542F17A1F4D8696638E401F258
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/79083E539F23B720B9EB08C589F4994B51C080EE
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/7F7AD323CF8ED99D434CBE3F180E446B2EC32E80
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/786F0FA8C5E63FF1D30132ABE76A364B1E389EE7
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/562A19DC080EA539452CE88F5CAAEB7A9A9B4B61
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/C915E09F433E78D4AA52F7A027B4ACA29F5902C7
4       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/82B7097C188FBD581F202219E1029AC31B12C9B1
36      ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/D2E971C56EED5650FA2E42C087B3043F21215402
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/AD38FCF996CD324FEAA19BE54D71715AE992FE9B
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/4E737D0A77B38CC470C5FF88084CCDD43A06664C
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/6FC6C3EE59456AAF56B5FE04EF52BBFDAB62C6A2
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/85E4DEDB12D4D6482A35A2656A298BBA0B7E34C6
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/0EF73CD26BD56FE57F251E8BFD24DCA3FAC909CC
8       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/AF503582175A576A4036FD460297E27EEA34E89D
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/7E2A5F8CAFCC8D41259616C11E29469CFDE4AF2D
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/E9C416A4D74500C933A0F3F03B6C99153CFE5165
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/6ADA4082BD7ABEFC339A5B6296BB7C19B4F65357
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/FB53CF7973B1130ACD0D191966153E4C6B0F227A
4       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/9899BB91461E965A287A6CE4BB5858289839AB04
24      ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/09C22C5DC2179AE58DB3688DB21734180E7BEBBB
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/21722B308B0C0687105CB2703BB91FE7DA5E1560
        ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/2C73272B7F5FFCA4D5B9800B6A126E7380BBEE89
4       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/C5E59F2D4F083DF4D36A1FC20D6293B92300B359
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/00DE55DEF4975BFC593F373A5BB92545BDBD6C8F
2       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/139E19A9883F7BB43622C8C2FA0030BC704AE95A
6       ./.cache/mozilla/firefox/bfx8q729.default-release/cache2/entries/944772DF35680D2FC5EE150A2678176C339F0518
```

- **du -s: -** This will show you the total disk space used by a specific directory or file.

  Syntax:

  $ du -s

```
preeti@javatpoint:~$ du -s
252348  .
```

- **du -help: -** we can use this for help.

  Syntax

  $ du -help

```
preeti@javatpoint:~$ du --help
Usage: du [OPTION]... [FILE]...
  or:  du [OPTION]... --files0-from=F
Summarize disk usage of the set of FILEs, recursively for directories.

Mandatory arguments to long options are mandatory for short options too.
  -0, --null            end each output line with NUL, not newline
  -a, --all             write counts for all files, not just directories
      --apparent-size   print apparent sizes, rather than disk usage; although
                          the apparent size is usually smaller, it may be
                          larger due to holes in ('sparse') files, internal
                          fragmentation, indirect blocks, and the like
  -B, --block-size=SIZE  scale sizes by SIZE before printing them; e.g.,
                           '-BM' prints sizes in units of 1,048,576 bytes;
                           see SIZE format below
  -b, --bytes           equivalent to '--apparent-size --block-size=1'
  -c, --total           produce a grand total
  -D, --dereference-args  dereference only symlinks that are listed on the
                          command line
  -d, --max-depth=N     print the total for a directory (or file, with --all)
                          only if it is N or fewer levels below the command
                          line argument;  --max-depth=0 is the same as
                          --summarize
      --files0-from=F   summarize disk usage of the
                          NUL-terminated file names specified in file F;
                          if F is -, then read names from standard input
  -H                    equivalent to --dereference-args (-D)
  -h, --human-readable  print sizes in human readable format (e.g., 1K 234M 2G)
      --inodes          list inode usage information instead of block usage
  -k                    like --block-size=1K
  -L, --dereference     dereference all symbolic links
  -l, --count-links     count sizes many times if hard linked
  -m                    like --block-size=1M
  -P, --no-dereference  don't follow any symbolic links (this is the default)
  -S, --separate-dirs   for directories do not include size of subdirectories
      --si              like -h, but use powers of 1000 not 1024
```

2. Using the df option

   The full-form of df command is "disk-free,". Using this command, we can check the used and available disk space in the Linux system.

   Syntax:

   $ df [optios].....FILESYSTEM.....

   When we use this command with no parameter, then this command will show you the information related to all the mounted file systems:

df Command in Linux/Unix with Examples

Unix/Linux df command is used to display the **disk space used in the file system**. The 'df' stands for **"disk file system**." It defines the number of blocks used, the number of blocks available, and the directory where the file system is mounted.

Syntax:

df [OPTION]... [FILE]...

Options:

-a, --all: It is used to include pseudo, duplicate, remote file systems.

-B, --block-size=SIZE: It is used to scale sizes by SIZE before printing them, for example, the '-BM' option prints sizes in units of 1,048,576 bytes.

-h, --human-readable: It is used to display sizes in powers of 1024 (e.g., 1023M).

-H, --si: It is used to show sizes in powers of 1000 (e.g., 1.1G)

-i, --inodes: It is used to list inode information instead of block usage

-l, --local: It is used to limit the listing to local file systems.

--no-sync: It is used for not invoking sync before getting usage info (default).

--output[=FIELD_LIST]: This option used if we want to use the output format defined by FIELD_LIST or print all fields if FIELD_LIST is omitted.

-P, --portability: It is used to use the POSIX output format.

--total: It is used to exclude all entries insignificant to available space, and produce a total.

-t, --type=TYPE: It is used to limit the listing to file systems of type TYPE.

-T, --print-type: It is used to display the file system type.

-x, --exclude-type=TYPE: It is used to limit the listing to file systems, not of type TYPE.

--help: It is used to display the help manual having brief information about the supported options.

--version: It is used to display the version information of the df command.

Examples of the df command

Let's see the following examples of the df command:

- Display the disk space usage
- Display the disk space usage in a human-readable form
- Display the file system type
- Display specific file system types
- Exclude the particular file system types
- Display available space and mount point for a folder

Display the Disk Space Usage

To display the disk space usage, execute the df command without any argument. It will show the disk space usage in a tabular form. The df command is useful for discovering the available free space on a system or file system. Execute the below command:

df

The above command will produce the output as follows:

```
javatpoint@javatpoint-Inspiron-3542:~$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             1931652        0   1931652   0% /dev
tmpfs             393260     1760    391500   1% /run
/dev/sda1      479668904 29005756 426227540   7% /
tmpfs            1966284   304388   1661896  16% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
tmpfs            1966284        0   1966284   0% /sys/fs/cgroup
/dev/loop1          1024     1024         0 100% /snap/gnome-logs/93
/dev/loop3          2560     2560         0 100% /snap/gnome-calculator/748
/dev/loop4          2560     2560         0 100% /snap/gnome-calculator/730
/dev/loop0          2304     2304         0 100% /snap/gnome-system-monitor/148
/dev/loop5           384      384         0 100% /snap/gnome-characters/550
/dev/loop8          1024     1024         0 100% /snap/gnome-logs/100
/dev/loop7        178048   178048         0 100% /snap/gimp/252
/dev/loop2        261760   261760         0 100% /snap/gnome-3-34-1804/33
/dev/loop9         56320    56320         0 100% /snap/core18/1705
/dev/loop6         99456    99456         0 100% /snap/core/9289
/dev/loop15       220160   220160         0 100% /snap/wine-platform-5-stable/5
/dev/loop13        96256    96256         0 100% /snap/core/9066
/dev/loop11        56192    56192         0 100% /snap/gtk-common-themes/1502
/dev/loop10       261760   261760         0 100% /snap/gnome-3-34-1804/36
/dev/loop12       180096   180096         0 100% /snap/gimp/273
```

From the above output, we can see the file system, the size of the file system in 1k block, used space, available space, the percentage applied by the file system, and mount point, respectively.

Display the disk space usage in a human-readable form

The '-h' option is used to display the disk space in a human-readable form. It will display the size in powers of 1024 and will append G for GBs, M for MBs, and B for Bytes. Execute the below command:

df -h

The above command will produce the output as follows:

```
javatpoint@javatpoint-Inspiron-3542:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            1.9G     0  1.9G   0% /dev
tmpfs           385M  1.8M  383M   1% /run
/dev/sda1       458G   28G  407G   7% /
tmpfs           1.9G  302M  1.6G  16% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           1.9G     0  1.9G   0% /sys/fs/cgroup
/dev/loop1      1.0M  1.0M     0 100% /snap/gnome-logs/93
/dev/loop3      2.5M  2.5M     0 100% /snap/gnome-calculator/748
/dev/loop4      2.5M  2.5M     0 100% /snap/gnome-calculator/730
/dev/loop0      2.3M  2.3M     0 100% /snap/gnome-system-monitor/148
/dev/loop5      384K  384K     0 100% /snap/gnome-characters/550
/dev/loop8      1.0M  1.0M     0 100% /snap/gnome-logs/100
/dev/loop7      174M  174M     0 100% /snap/gimp/252
/dev/loop2      256M  256M     0 100% /snap/gnome-3-34-1804/33
/dev/loop9       55M   55M     0 100% /snap/core18/1705
/dev/loop6       98M   98M     0 100% /snap/core/9289
/dev/loop15     215M  215M     0 100% /snap/wine-platform-5-stable/5
/dev/loop13      94M   94M     0 100% /snap/core/9066
/dev/loop11      55M   55M     0 100% /snap/gtk-common-themes/1502
/dev/loop10     256M  256M     0 100% /snap/gnome-3-34-1804/36
/dev/loop12     176M  176M     0 100% /snap/gimp/273
/dev/loop14     141M  141M     0 100% /snap/gnome-3-26-1604/100
```

Display the file system type

The '-T' option is used to display the file system type. It will add a new column having the file system type to output. Execute the below command:

df -T

The above command will produce the output as follows:

```
javatpoint@javatpoint-Inspiron-3542:~$ df -T
Filesystem     Type     1K-blocks     Used Available Use% Mounted on
udev           devtmpfs   1931652        0   1931652   0% /dev
tmpfs          tmpfs       393260     1768    391492   1% /run
/dev/sda1      ext4     479668904 29007564 426225732   7% /
tmpfs          tmpfs      1966284   300776   1665508  16% /dev/shm
tmpfs          tmpfs         5120        4      5116   1% /run/lock
tmpfs          tmpfs      1966284        0   1966284   0% /sys/fs/cgroup
/dev/loop1     squashfs      1024     1024         0 100% /snap/gnome-logs/93
/dev/loop3     squashfs      2560     2560         0 100% /snap/gnome-calculator
/748
/dev/loop4     squashfs      2560     2560         0 100% /snap/gnome-calculator
/730
/dev/loop0     squashfs      2304     2304         0 100% /snap/gnome-system-mon
itor/148
/dev/loop5     squashfs       384      384         0 100% /snap/gnome-characters
/550
/dev/loop8     squashfs      1024     1024         0 100% /snap/gnome-logs/100
/dev/loop7     squashfs    178048   178048         0 100% /snap/gimp/252
/dev/loop2     squashfs    261760   261760         0 100% /snap/gnome-3-34-1804/
33
/dev/loop9     squashfs     56320    56320         0 100% /snap/core18/1705
/dev/loop6     squashfs     99456    99456         0 100% /snap/core/9289
```

From the above output, we can see the column 'type' has been added with output.

Display specific file system types

The '-t' option is used with the file system type to display the specific file system. It will only display a given file system. We can specify more than one file system with it. Consider the below command:

df -t ext4

the above command will only display the 'ext4' types file system. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ df -t ext4
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1      479668904 29009036 426224260   7% /
javatpoint@javatpoint-Inspiron-3542:~$
```

Exclude the specific file system types

The '-x' option is used with the specific file system type to exclude it from the output. It will display all other file system types except the given types. Consider the below command:

df -x squashfs

The above command will exclude the 'squashfs' file system from the output. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ df -x squashfs
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             1931652        0   1931652   0% /dev
tmpfs             393260     1828    391432   1% /run
/dev/sda1      479668904 29011808 426221488   7% /
tmpfs            1966284   303156   1663128  16% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
tmpfs            1966284        0   1966284   0% /sys/fs/cgroup
tmpfs             393256       68    393188   1% /run/user/1000
```

Display available space and mount point for a folder

To display the available space, file system type, and mount point of a folder, pass the folder name with the df command. Consider the below command:

df Newdirectory

The above command will display the file system details of the given folder. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ df Newdirectory
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1      479668904 28999676 426233620   7% /
```

From the above output, we can see the used disk space, file system types, and other information of the given folder are displayed. We can make the output more specific by passing the supported options.

3.9 UNLIMIT COMMAND

ulimit is admin access required Linux shell command which is used to see, set, or limit the resource usage of the current user. It is used to return the number of open file descriptors for each process. It is also used to set restrictions on the resources used by a process.

Syntax:

To check the ulimit value use the following command:

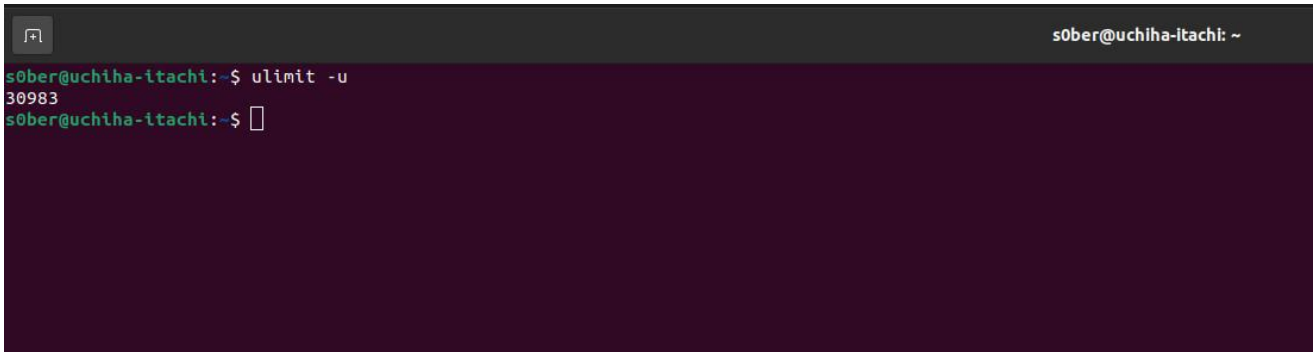
ulimit -a

```
s0ber@uchiha-itachi:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 30982
max locked memory       (kbytes, -l) 998850
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 30982
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
s0ber@uchiha-itachi:~$
```

Working with ulimit commands:

1. To display maximum users process or for showing maximum user process limit for the logged-in user.

   ulimit -u



2. For showing the maximum file size a user can have.

   ulimit -f

3. For showing maximum memory size for the current user.

   ulimit -m

4. For showing maximum memory size limit.

   ulimit -v

What are Soft limits and Hard limits in Linux?

The soft limits are the limits which are allocated for actual processing of application or users while the Hard limits are nothing but an upper bound to the values of soft limits. Hence,

(soft limits <= hard limit)

Working with Hard and Soft limit values:

1. For displaying the Hard limit. Hard limits are a restriction to the maximum value of soft limits

   ulimit -Hn

2. For displaying Soft Limit. The soft limits are the limits that are there for processing.

   ulimit -Sn

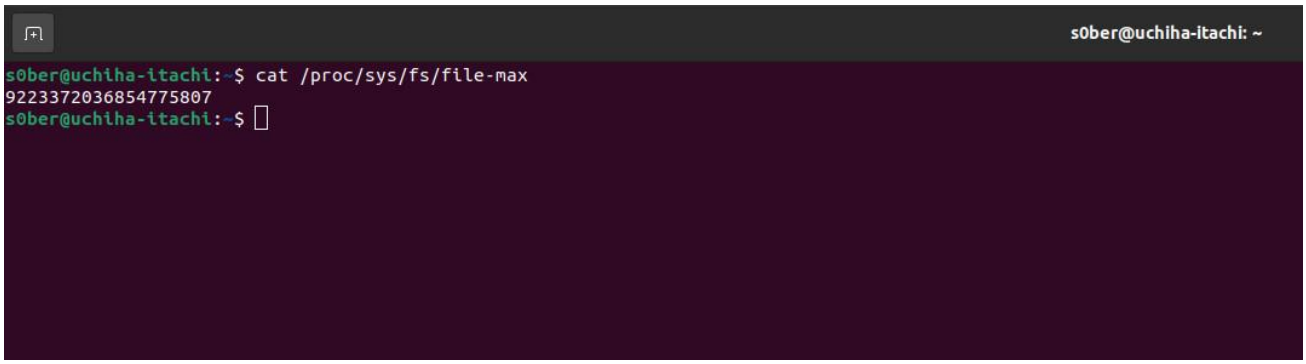
3. To change Soft Limit values:

   sysctl -w fs.file-max=<value>

Note: Replace <value> with the value you want to set for the soft limit and also remember size can not exceed the Hard Limit!

4. Displaying current Values for opened files

cat /proc/sys/fs/file-max



3.10 SUMMARY

In this unit, we have learn about the file system and types of file block, Boot block, super block, Inode table, data blocks, How Unix access files, storage files, Disk related commands: checking disk free space, dfsapce, du, ulimit etc

3.10 CHECK YOUR PROGRESS

- **What do you mean by types file block?**

……
……
……
……
……
……

- **What do mean by the du and df command?**

……
……
……
……
……
……

- **Discuss how how access file in unix.**

……
……
……
……
……
……
……

- **What is unlimit command**

  ……
  ……
  ……
  ……
  ……

Further Reading:

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education

UNIT-4 ESSENTIAL COMMANDS OF UNIX

Structure

4.1 INTRODUCTION

An OS providing both command line interface (CLI) and graphical user interface (GUI) based interaction integrated by Dennis Ritchie, Douglas Mcllroy, Joe Ossanna Brian Kernighan, and Ken Thompson at Bell laboratory in 1970 called a multi-tasking operating system permitting two or more users to simultaneously operate on the operating system and offers commands for the users for interacting with the application from command line interface, such as sudo command, chmod command, su command, mv command, rm command, vi command, cat command, rmdir command, mkdir command, clear command, and ls command, which can be used to implement complex task

Unix is an OS that provides both CLI and GUI-based interaction. It was developed by Dennis Ritchie in the C language. Unix operating system is multitasking, which also gives an opportunity for two or more users to use its benefits. In other words, it is a multi-user OS. Ubuntu OS is a Unix version that enables us to do every work that Unix is supposed to do.

Several testing activities, such as performance and installation testing, depending on OS knowledge. Almost every web server is Unix based nowadays. So, knowledge of Unix is essential for testers. If

we are unfamiliar with Unix, then learning Unix commands can be a great start. One of the best ways to understand these commands is to practice and read them simultaneously on Unix OS.

4.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- Concept of Unix commands
- Know about **Types of command**,
- Learn about Commands for files and directories like cd, ls, cp, md, rm, mkdir, rmdir, more, less, creating and viewing files, using cat, file comparisons, file compression commands,
- Learn about unix manual page

4.2 UNIX COMMAND

Unix has large number of commands. Types of Unix commands are two types .

1. External commands
2. Internal commands

External commands: A command with an independent existence in the form of a separate file is called an external command. Ex: cat and ls . these are independently existed in a directory called the / bin directory . When external commands and given the shell reaches these command files with the help of 'PATH' variable.

Internal commands: A command that does not have an independent existence is called an internal command. Ex. Echo,mkdir,cd,etc; The routines for internal commands will be a part of another program (or) routine. Ex. Echo command is internal command will be a part of another program(or) routine is the part of shell's routine "sh".

Some basic commands:- Unix has several hundreds of commands with it. some of them are.

1. echo
2. tput
3. lty
4. who
5. uname
6. date
7. cal
8. calendar
9. passwd
10. lock
11. banner
12. cat etc..;

The echo command:-

Use:-

- The echo command is used to display messages.
- It takes zero,one(or) more no of arguments.
- Arguments may be given either as a series of individual symbols or as a string with in a pair of double quotes(“ “).
- Example:

1. $echo #Blank line is displayed

   $

2. $echo Iam studying information technology I am studying information technology

   $

3. $echo “Iam studying unix programming”. Iam studying unix programming.

The tput command:-

Use:-

- This command is used to control movement of the cursor on the screen as well as to add the certain features like blinking ,boldface and underlining to the displayed message on the screen.
- Examples :-
  - $tput clear
- Clears the screen and puts the cursor at the left top of the screen
  - $tput cup 10 20
- This command along with cup a argument and certain co-ordinates values is used to position the cursor at any required position on the screen.
  - In the above example the cursor will be placed at the tenth and twentieth column on the screen.
  - $tput lines

    48

    $
- To know the current no of rows on terminal.
  - $tput cols

    142

    $

The tty command:-

In unix, every terminal is associated with a special file, called the device file. All the device files will be present in /dev directory.

Use:- Examples:

- $tty /dev/tty 01

$

- Here tty01 is the file name and will be available in /dev directory.

The who command:-

Use:-

The user can know the login details of the current users by using the who command .

Ex:- $who

Root console nov 19 09:35

Mgv tty01 nov 19 09:40

dvm tty02 nov 19 09:45

- The first column shows the name of the users,the second column shows devicename and third column shows login time.

  Ex:- $who am i

  Mgv tty01 nov 19 9:40

- The self-login details of a user can be obtained in the above example

The uname command:-

Use: -

- using this command one can know the details of one's unix system.
- Unix has many variants, when this command is used .it gives the name of Unix system being used by the user.

  Ex:- $uname

  Linux

  $

The date command:-

- Using this command, the user can display the current date along with the time nearest to the second.

  Ex:-

  - $date Sat jan 18 11:58:00 lst 2018

    $

  - $date +%m

  09

  $

- Displays only month in numeric form.

  - %date +%h

  Sep

$

- Displays the name of the month

The cal command:- Use:-

- This command is used to print the calendar of a specific month or a specific year.
- When this command is used without any arguments the calendar of the current month of the current year will be printed.

Ex:-

$cal 02 2018

Jan 2018

Su	mo	tue	wed	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	31	

$

The passwd command:-

- As unix is a multi-user system due to which there is always a security threat.
- The simplest and most widely used by all individual users is use of passwds.
- For every new user,the administrator ,permita or authorities them by assigning unique password to each.

Use:-

- A user can change his/her password by using this command.

Ex:- $passwd: *****

Old passwd: *******

New passwd: *******

$

When a user attempts to change his/her password the system first asks for old password ,if it is correctly entered the system asks twice to enter new password and srts it as current password

The lock command:-

Use:-

- The lock command is used for locking a session for any required amount of time.

- This command is used generally without any arguments.
- By default the user can lock it for 30 min.
- The locking period can be changed by assigning a different value forthe system variable DEFLOGOUT.
- When the lock command is given, the terminal asks for a password twice.

Eg: $lock -45 #locks for 45 min

The banner command:

- It is used to display banners or pasters.
- This command simply produces a blowup version of the characters that are supplied with the command.
- Ex: $ banner hello

The cat command:

- This is used to create small unix files. A file can be created by writing a command line.
- Eg: $ cat > review

A > symbol following the command

Means that the output

Goes to the file name following it

<ctrl -d>

- In the above example, 'review' is the file name after executing $cat > review, the $ disappears and the presents '>' symbol which means the system is ready to accept the contents to be placed in the file.
- If the file contents are over to exit, we have to use the command <ctrl -d>

The bc command:

- The bc command is both a calculator and a small language for writingnumerical programs.
- By using this one can perform all the usual arithmetic operations as well as change of bases in the range of 2 -16
- Ex: $ bc

Sqrt 55

7 Quit

$

The spell and ispell command:

- The spell command is the first program that was developed to check for words that are wrongly spelt in a document.
- This command displays a list of misspelled words in the document used as argument.
- Eg: $cat spell – ux

This is example

I am using cat command

*

$ spell spell

. ux Example

Command

$

Ispell:

The ispell command is an interactive spell – check program available in linux . when used, this command displays a screen full of information in tree sections.

Eg: $ispell spell.ux

This is example I am using cat command

1. Example
2. Exempler
3. Exampled

I.	Ignore	ignore all
II.	Replace	replace all
III.	Add	exit

The man command:

The man command prints the details of any command of a utility on the screen.

Ex: $man pwd

Pwd(1) user commands pwd(1)

Name

Pwd - print working directory pwd(1)

Synopsis

Pwd

Description

Pwd prints the pathname of the working directory Notes

Command substitution:

In unix, it is possible to run a command within another command.

Ex: The date command can be run within the echo command by writing command as

$ echo today the date is ‘date’

Today the date is the june 7 16:25:00 2024

when more trend is executed the shell while parsing the parameters list of the echo command treats the words that are back quoted as a command executes it and substitutes the result of this execution at the corresponding position in the parameter list. This process is known as command substitution.

4.3 MORE ABOUT COMMAND

All programs in Unix are accessed by typing the program’s name into a command shell. When you open a command shell you should see the last line looks something like this:

[andrewss@rocks1 ~]$

If you wanted to run the “whoami” command to find out what your username is you’d simply type that after the $ and press return.

```
[andrewss@rocks1 ~]$ whoami
andrewss
```

The system will search through a set of directories, called the PATH, to try to find a match to the command name you gave. If it can’t find a match you’ll get an error like this:

```
[andrewss@rocks1 ~]$ whoareyou
-bash: whoareyou: command not found
```

All operations in Unix are case-sensitive so you need to get the case as well as the command name correct. In general all core Unix commands are all lower case.

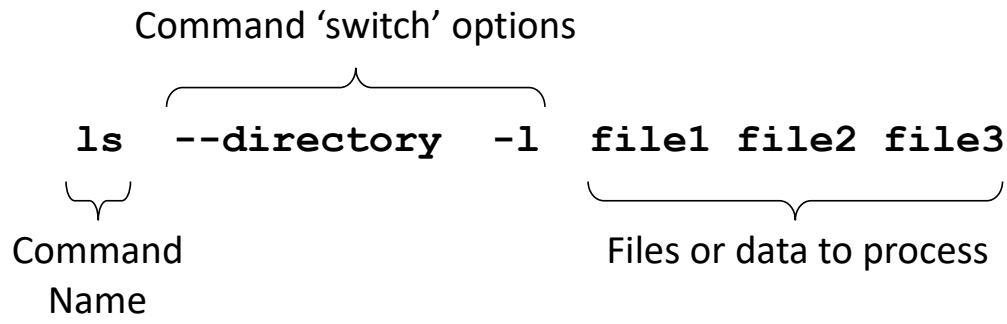
```
[andrewss@rocks1 ~]$ whoami
andrewss
[andrewss@rocks1 ~]$ WhoAmI
-bash: WhoAmI: command not found
```

If you want to run a program which isn’t in your PATH then you can do this by specifying the full location of the program rather than having the shell search for it. This also applies to running a program in your current directory where you’d need to tell the shell it was OK to run that program even though it wasn’t in your PATH

```
[andrewss@rocks1 ~]$ /bi/home/andrewss/whoareyou
You are number 3
[andrewss@rocks1 ~]$ ./whoareyou
You are number 3
```

Unix commands generally require you to provide a series of options along with the command name so that the program knows exactly what to do.

Additional information is provided either by adding command 'switches' after the program name or by providing data to process (usually lists of file names) at the end. The typical anatomy of a Unix command is therefore something like:



Command switches are generally specified by either having two dashes and a long name, or a single dash and a single letter name. For a specific option there are often both single letter and long option variants, so -d and --directory might do the same thing.

For single letter command switches you can join the options together in a single set of option letters to save space, so instead of doing:

```
-l -t -r
```

You could simply do

```
-ltr
```

Although some command options tend to get reused by a lot of programs there is no enforced standard for what options are provided to any program, so the only way to be sure is to look at its documentation. For most core Unix commands you can see the documentation for it by looking at the manual page. You can access this by running:

```
man [command name]
```

Man pages tend to be fairly technical, but they will always list all of the command switches and what they do. They will also say what other data needs to be passed after the switches. The start of the man page for ls looks like this for example:

LS(1) User Commands LS(1)

NAME

ls - list directory contents

Syntax:

ls [OPTION]... [FILE]...

DESCRIPTION

List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.

-a, --all

do not ignore entries starting with .

-A, --almost-all

do not list implied . and ..

--author

with -l, print the author of each file

-b, --escape

print octal escapes for nongraphic characters

If you find that the program you're using doesn't have a man page (many scientific applications won't have one for example), then you can normally get at the programs help by adding the --help or -h command switch.

[andrewss@rocks1 ~]$ fastqc --help

FastQC - A high throughput sequence QC analysis tool

4.4 FILES AND DIRECTORIES

All of your file and directory management in a Unix environment is generally achieved using command line programs. There are graphical file managers, but it's a good idea to learn how to manage your files using the command line, since many of the operations you learn in doing this will be of use when processing data using other command line programs later on.

4.5 FILE PATHS

Unix uses a file system based on a tree of directories. When you see the full path to a directory it will be shown as a list of directory names starting from the top of the tree right the way down to the level where your directory sits. Directory names in a path are separated by a / (forward slash) character. A typical directory name might be something like:

/home/andrewss/data/february/RNA-Seq

Directory names can contain spaces but you'll often find that people working in Unix prefer to keep both file and directory names without spaces since this simplifies the process of working with these files on a command line. Directory or file names containing a space must be specified on the command line either by putting quotes around them, or by putting a \ (backslash) before the space.

"/home/andrewss/Directory Containing Spaces"

/home/andrewss/Directory\ Containing\ Spaces

Unix does not have separate drives as you might see under windows, but instead can attach different storage devices at any point in the directory tree, so you could add in a new disk and put that at /home so all directories under /home go onto that disk. When using the filesystem you generally don't need to worry about where physical disks or partitions exist in the file tree.

When specifying directory paths you can do this either by constructing a full path from the top of the directory tree (called the root directory or /), or you can construct a relative path from your current location. To help in constructing paths there are some directory aliases which you can use:

. (single dot) means the current directory

.. (double dot) means the directory immediately above the current directory

~ (tilde) means the home directory for your user

So you can make directory paths like this:

- /data/projects/simon/analysis/ - a full directory path to a specific location
- projects/analysis/ - a directory called analysis which is in a directory called projects which is in the current directory. This could also be written ./projects/analysis
- ../../analysis/ - go up two directory levels from the current directory and then down into a folder called analysis
- ~/scripts/ - a directory called scripts which is inside your home directory

One additional convention you might see is that any file or directory whose name starts with a dot is classified as a hidden directory and will not show up in normal file listings. You should therefore avoid using a dot at the start of a file or directory name unless you want it to be hidden.

Whilst you can type out full directory or file paths, a really helpful tool on the command line is tab completion. If you start typing a file path you can press the tab key - if the part you have typed only matches one possible file or directory then the rest of the name is filled in automatically. If it can match more than one completion then nothing will happen, however if you press tab again the shell will show you what possible completions there could have been so you can see what options you have.

$ /bi/home/andrewss/Sc [hit tab]

$ /bi/home/andrewss/Scripts/ (automatically completes the name)

[hit tab] - nothing happens

[hit tab again]

Bash/ Perl/

$ /bi/home/andrewss/Scripts/

Shows the possible completions and restores the command so you can continue it.

4.6 LISTING FILES AND DIRECTORIES

To see what files are already on your file system you can use the ls command. If you run this without any arguments it will provide a simple listing of the contents of the current directory. If you pass it a specific directory path it will list the contents of that directory instead.

Common options you'll use with ls are:

- ls -l List in long format which shows the permissions, ownership, modification date and size of files in addition to their name. File sizes are shown in bytes, but you can add the -h flag to get them in more human friendly units (ie ls -lh)
- ls -a Lists files but includes files whose name starts with a . and which would normally be hidden
- ls -d When listing a specific directory lists the directory itself and not its contents.

4.7 CREATING DIRECTORIES

The command to create a directory is mkdir. If you simply pass it a new name it will create a directory with that name in your current location. You can also pass it a full path to the new directory you want to create but this will only work if only the last item in the path needs to be created. If you want to create a set of directories all at once you need to pass the -p flag.

$ ls -l

total 0

$ mkdir new_directory

$ ls -l

total 2

drwxrwxr-x 2 andrewss bioinf 0 Oct 11 11:36 new_directory

$ mkdir another_dir/with_subdir

mkdir: cannot create directory `another_dir/with_subdir': No such file or directory

$ mkdir -p another_dir/with_subdir

$ ls -l

total 4

drwxrwxr-x 3 andrewss bioinf 29 Oct 11 11:36 another_dir

drwxrwxr-x 2 andrewss bioinf 0 Oct 11 11:36 new_directory

$ ls -l another_dir/

total 2

drwxrwxr-x 2 andrewss bioinf 0 Oct 11 11:36 with_subdir

4.8 CHANGING DIRECTORY

When working in Unix your command shell has one directory which it is using as its working directory, and all file paths you use start in that directory unless you specifically say otherwise. When you first log in your working directory will be your home directory, but you can change this to wherever you actually want to do your work.

You can find your current working directory path using the pwd command. You will also see that the name (not the path) of your current directory will also be shown in your command prompt.

[andrewss@rocks1 bioinf]$ pwd

/bi/group/bioinf

You can change your working directory using the cd (change directory) command. You pass this either a relative or an absolute path to a new directory and your working directory will be moved to that location.

$ pwd

/bi/group/bioinf

$ cd Steven/

$ pwd

/bi/group/bioinf/Steven

$ cd ../..

$ pwd

/bi/group

$ cd ~

$ pwd

/bi/home/andrewss

$ cd /bi/scratch/Genomes/

$ pwd

/bi/scratch/Genomes

4.9 WILDCARDS

When specifying a file or directory you often want to generate a list of several related files (all the files in the current directory ending in .txt for example). Unix provides an easy way to do this using wildcards.

A wildcard is simply part of a file or directory path which can match multiple pieces of text. There are actually quite a large number of ways of specifying a wild card, but the two common ones which will work for most situations are:

1. * This matches any number of any character (including the empty string)
2. ? This matches exactly one instance of any character.

You can construct file or directory paths using multiple wildcards and your command shell will substitute these with the matching set of real file or directory paths before running the command, so from the point of view of the command it will look as if you actually typed out all of the individual files.

If I wanted to list all of the Perl (.pl) files in the current directory I could therefore do:

$ ls *pl

copy_bristol_data.pl make_fastq_from_raw.pl remove_spaces.pl

extract_tiles.pl remove_blank_lines.pl rename_tophat_bam_files.pl

I can also use multiple wildcards - if I wanted to list all text files (.txt) in any subdirectory of the current directory I can do:

```
$ ls */*txt
bin/CREATING_CUSTOM_GENOMES.txt      FastQC/RELEASE_NOTES.txt
bin/INSTALL.txt                      group/file_list.txt
bin/LICENSE.txt                      SeqMonk/CREATING_CUSTOM_GENOMES.txt
bin/NOTICE_FOR_COMMONS_MATH_CODE.txt SeqMonk/INSTALL.txt
bin/README.txt                       SeqMonk/LICENSE.txt
bin/RELEASE_NOTES.txt                SeqMonk/NOTICE_FOR_MATH_CODE.txt
FastQC/INSTALL.txt                   SeqMonk/README.txt
FastQC/LICENSE.txt                   SeqMonk/RELEASE_NOTES.txt
FastQC/README.txt                    subread-1.3.5-p5/README.txt
```

One word of warning (especially when using wildcards when deleting!). Be very careful to ensure that you don't accidentally leave a space between a wildcard and the rest of your string. If instead of doing:

```
ls *pl
```

You did

```
ls * pl
```

You actually create two file lists, one from * and one from pl. * on its own will list everything in the current directory and unless you have a file called pl the second won't match anything. If you had linked this wildcard to a delete operation this is a really easy way to wipe out lots of files.

4.10 VIEWING FILES

Most of the data files you'll be working with on a Unix system will be simple text files which you can look at within your command shell. You can put the whole contents of a file onto your screen using the cat command.

```
$ cat test.seq
>test
ATGATGGAA
```

Many files will be too big to fit onto a single command window so to use these you can use a paging program which will allow to scroll through a large file to view it in a more sensible manner. The most common paging program is called less.

```
$ less fastq_screen.conf
# This is a configuration file for fastq_screen

############
## Bowtie #
############
## If the bowtie binary is not in your PATH then you can
## set this value to tell the program where to find it.
## Uncomment the line below and set the appropriate location
##

#BOWTIE /usr/local/bin/bowtie/bowtie
#BOWTIE2 /usr/local/bowtie2/bowtie2

fastq_screen.conf
```

When less is running you can use the following keyboard shortcuts to move around:

- [space] go forward one page
- b go back one page
- j go forward one line
- k go back one line
- /[some text] search for the next occurrence of [some text]
- q exit.

4.11 TEXT EDITORS

There are many different text editors on Unix systems and there is no single editor which everyone has settled on as being the recommended one to use. The two most commonly used editors are vi and emacs but both of these are fairly intimidating to new users and have some very unusual features

in the way that they operate, but they are both very powerful and are both likely to be found on most Unix systems (vi is guaranteed to be there). There are many introductory guides to these editors and the specifics of their use is beyond the scope of this manual but it would be a good idea to become familiar with one of these at some point.

For simple text editing there is a simple editor called nano which you can launch by typing:

nano some_file.txt

This will edit some_file.txt, and will create it if it doesn't exist. It has a very simple text based interface but it's generally functional and will do basic editing for you.

GNU nano 2.0.9 File: test.txt Modified

This is a simple text file which I can edit using simple controls.

The commands listed at the bottom are all run by pressing Control

plus the listed character, so Control+O writes out the text and

Control+X exits.

^G Get Help ^O **WriteOut** **^R** Read File ^Y Prev Page ^K Cut Text ^C Cur Pos

^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

4.12 COMMAND REDIRECTION

Another common way to create files in Unix is by sending the output of a command to a file. Some commands will explicitly ask for a location to save results, but many will simply write the results to their standard output location which will normally go to the screen, but can be sent to a file.

Output redirection is achieved using the > symbol. You simply add this after the command and any normal output will get send to the file name you specify immediately afterwards.

[andrewss@rocks1 group]$ ls -l > file_list.txt

[andrewss@rocks1 group]$ cat file_list.txt

total 1054

drwxrwxr-x 4 andrewss bioinf 69 Apr 22 12:29 Alison_Galloway

drwxrwxr-x 6 fkrueger bioinf 765 Jun 3 14:19 Andrew_Dimond

drwxrwxr-x 5 andrewss bioinf 121 Oct 10 10:45 Cameron_Osborne

drwxrwxr-x 6 ewelsp bioinf 100 Feb 22 2013 Chandra_2012

drwxrwxr-x 3 andrewss bioinf 36 Jun 17 12:17 Claire_Dawson

drwxrwxr-x 2 fkrueger bioinf 234 Apr 30 16:22 Claire_Senner

drwxrwxr-x 6 ewelsp bioinf 189 Sep 25 13:29 clusterflow

drwxrwxr-x 4 ewelsp bioinf 46 Feb 21 2013 Dawlaty_2013

Using > will create a new file, or if the file exists it will wipe the existing contents and start afresh. If you want to append content to an existing file you can use >> instead.

All programs actually have two types of output they can produce, standard output (often called stdout) is for expected output, and standard error (stderr) is for progress messages, warnings and errors. Normal redirection only redirects stdout. If you want to redirect stderr you can either send this to a different file using 2> [error file] or you can mix it in with the standard output using 2>&1.

4.13 MOVING AND RENAMING

In Unix the process to rename a file or directory to a new name and the process of moving it to a different place in the filesystem are the same. They are both achieved using the mv command, which requires you to provide the original name and location of the file and the new name and location you want to use.

$ ls

a_file subdir

$ mv a_file b_file

$ ls

b_file subdir

$ mv b_file subdir/

$ ls subdir/

b_file

If the location to move to is a directory which already exists then the file will be moved into that directory under the same name.

Deleting

Deletion of files and directories is most commonly achieved with the rm (remove) command. This takes in a list of files and removes them. If the -r flag is specified then the remove is 'recursive' and this will remove any directories listed and their contents. The rm command does not prompt you for any confirmation and with the -r flag it can potentially delete large portions of your file tree, so be very careful when using this command.

$ ls -l

total 4

-rw-rw-r-- 1 andrewss bioinf 0 Oct 11 12:53 a_file

drwxrwxr-x 2 andrewss bioinf 24 Oct 11 12:49 subdir

[andrewss@rocks1 Mkdir_test]$ rm a_file

[andrewss@rocks1 Mkdir_test]$ ls -l

total 2

drwxrwxr-x 2 andrewss bioinf 24 Oct 11 12:49 subdir

[andrewss@rocks1 Mkdir_test]$ rm subdir/

rm: cannot remove `subdir/': Is a directory

[andrewss@rocks1 Mkdir_test]$ rm -r subdir/

[andrewss@rocks1 Mkdir_test]$ ls -l

total 0

4.14 FILE COMPARISON COMMAND

The file comparison command helps us to compare the files and find the similarities and differences between these files. The different file comparison commands used in Unix are cmp, comm, diff, dircmp, and uniq.

Different ways of comparing two files in Unix

1) **cmp**: This command is used to compare two files character by character.

Syntax: cmp [options] file1 file2

Example: Add write permission for user, group and others for file1.

- *$ cmp file1 file2*

2) **comm**: This command is used to compare two sorted files.

Syntax: c*omm [options] file1 file2*

One set of options allows selection of 'columns' to suppress.

-1: suppress lines unique to file1 (column 1)

-2: suppress lines unique to file2 (column 2)

-3: suppress lines common to file1 and file2 (column3)

Example: Only show column-3 that contains lines common between file1 and file2

$ comm -12 file1 file2

3) **diff**: This command is used to compare two files line by line.

Description: The output indicates how the lines in each file are different, and the steps invoved to change file1 to file2. The 'patch' command can be used to make the suggested changes. The output is formatted as blocks of:

Change commands

< lines from file1—

> lines from file2

The change commands are in the format [range][acd][range]. The range on the left may be a line number or a comma-separated range of line numbers referring to file1, and the range on the right similarly refers to file2. The character in the middle indicates the action i.e. add, change or delete.

- 'LaR' – Add lines in range 'R' from file2 after line 'L' in file1.
- 'FcT' – Change lines in range 'F' of file1 to lines in range 'T' of file2.
- 'RdL' – Delete lines in range 'R' from file1 that would have appeared at line 'L' in file2
- **Syntax**: *diff [options] file1 file2*

Example: Add write permission for user, group and others for file1

$ diff file1 file2

4) **dircmp**: This command is used to compare the contents of directories.

Description: This command works on older versions of Unix. In order to compare the directories in the newer versions of Unix, we can use diff -r

Syntax: *dircmp [options] dir1 dir2*

- **Example**: Compare contents of dir1 and dir2

$ dircmp dir1 dir2

5) **uniq**: This command is used to filter the repeated lines in a file which are adjacent to each other

Syntax: *uniq [options] [input [output]]*

Example: Omit repeated lines which are adjacent to each other in file1 and print the repeated lines only once

$ uniq file1

4.15 FILE COMPRESSISON COMMAND

Websites will often allow you to download a single compressed file that contains many other files. This makes it easier and faster to download through your browser since the file is smaller than the folder containing all the files. These compressed files usually end with these extensions:

- .zip
- .gz
- .tar.gz
- .tar.bz2

Compressing a directory

Compressing to a .zip file

You can use the 'zip' command to compress a folder full of files. For example, if you have a WordPress site named **example.com**, you may want to compress all files and folders within it before downloading.

The following command compresses the directory named **example.com** and creates a new zip file named **example.com.zip**. The original directory remains untouched.

```
[server]$ zip -r example.com.zip example.com
```

You can use the maximum compression setting (-9) which will attempt to compress all files. However, this may be slower:

```
[server]$ zip -9 -r example.com.zip example.com
```

Compressing to a .tar.gz file

The following command uses 'tar' to compress an **images** directory into a file named **image_backup.tar.gz**.

```
[server]$ tar zcvf image_backup.tar.gz images
```

The original folder will continue to exist.

Compressing to a .tar.bz2 file

The following command uses 'tar' to compress an **images** directory into a file named **image_backup.tar.bz2**.

```
[server]$ tar cjvf image_backup.tar.bz2 images
```

The original folder will continue to exist.

Compressing files

Compressing a file using GZIP

The commands above show how to compress an entire directory. However sometimes you'll only need to compress a single file. To accomplish this you can use gzip. The following command compresses a file named **test.log** and renames it **test.log.gz**. Please note that the **test.log** file will no longer exist since it's been compressed.

```
[server]$ gzip test.log
```

If you still need the **test.log** file to exist, run the following instead. This creates a gzipped copy of the file:

```
[server]$ gzip -c test.log > test.log.gz
```

Confirming your folder has been compressed

Once you run the commands above, check to confirm the directory has been compressed. You can do this using `du -sh` for both the old and new directories.

```
[server]$ du -sh example.com.zip
30M example.com_backup
[server]$ du -sh example.com
82M example.com
```

You can see above that the compressed backup is smaller in size than the original folder.

What command compresses a directory the most?

The following shows how much a WordPress site is compressed using the different compression options.

First, check the size of the WordPress directory. This shows it's 82M.

```
[server]$ du -sh example.com
82M example.com
```

The following are the results of the different compression methods.

```
[server]$ du -sh example.com.zip
30M example.com.zip
[server]$ du -sh example.com.tar.gz
27M example.com.tar.gz
[server]$ du -sh example.com.tar.bz2
24M example.com.tar.bz2
```

Overall, tar.bz2 compresses the most.

4.16 MANUAL PAGES

Almost all Unix programs come with manual pages. These are terse, technical explanations of what the program does.To read a manual page for any command type man commandname (substituting the appropriate command name, of course).

For instance:

% man passwd

This will give you information about the passwd command which we used earlier. Here are the first few lines of the passwd man page:

PASSWD(1) PASSWD(1)

NAME

passwd - change user password

SYNOPSIS

passwd [-f|-s] [name]

passwd [-g] [-r|R] group

passwd [-x max] [-n min] [-w warn] [-i inact] name

passwd {-l|-u|-d|-S} name

DESCRIPTION

passwd changes passwords for user and group accounts. A normal user may only change the password for their own account, the super user may change the password for any account. The administrator of a group may change the password for the group. passwd also changes account information, such as the full name of the user, their login shell, or password expiry dates and intervals.

The man pages are presented to you using a pager program which allows you to view the text a screenful at a time. On most modern systems, this pager is a program called less. You can scroll through the man pages as follows:

Table 5-1. Navigating man pages

Movement	Keystroke
Down a page	SPACE
Up a page	b
Down one line	j or ENTER
Up one line	k
Search for a string	/string
Quit	q

Note: The above table assumes that your system uses the less program as its pager. Some older systems use more which does not allow you to go back in the file, only forward. An even more antiquated pager called pg is very occasionally seen; if your system uses this, be very afraid.

Finding the right man page:

If you have some idea what you want to do, but aren't sure of the exact command name, you can get a list of likely manual pages by using the apropos or man -k commands. These commands are roughly equivalent; both search for keywords in the single-line description of utilities and programs as given in their manual pages.

% man -k "card game"

canfield (6) - the solitaire card game canfield

cribbage (6) - the card game cribbage

% apropos "card game"

canfield (6) - the solitaire card game canfield

cribbage (6) - the card game cribbage

Categories for man pages

The manual pages on any Unix system are separated into categories. These categories vary slightly between systems, but typically they are:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within system libraries
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Macro packages and conventions eg man(7), groff(7).
8. System administration commands (usually only for root)

Sometimes it will happen that there are multiple manual pages with the same name, but in different sections of the manual. For instance, there is a manual page for the passwd command and also for the file with the same name which is found in /etc/passwd.You can distinguish between these manual pages by typing man n command where n is the section/category number.

% man 5 passwd

PASSWD(5) PASSWD(5)

NAME

passwd - The password file

DESCRIPTION

passwd contains various pieces of information for each user account. Included is

Login name

Optional encrypted password

...

As you can see, this is a different manual page to the one you would see if you just typed man passwd. The system will, by default, choose the lowest-numbered manual section first, so the man passwd would default to man 1 passwd (the passwd command) over the file /etc/passwd which is documented in section 5, and would be viewed by typing man 5 passwd.

Note: Some systems require you to use man -s section-number command instead of just man section-number command.

4.17 SYSTEM START UP AND SHUTDOWN

Start Up On Main Console

1. Press the Server main power button ON
2. The system will automatically go through the start-up procedures, which will be displayed on the main console screen.
3. Wait until it displays the word boot:
4. Press <Enter>or<return> and follow the steps shown below.

Type Control-D to proceed with norm at startup or enter the root password for system maintenance.

Hold down the control key at the same time press d

This will bring the system up in multiuser mode ,in normal working mode the screen will then display the date and time for correction or press Enter or return to accept

Current system time is……

Enter the new time……..

5. You may now turn on other Terminals. If the screens are blank, press or until the login prompt displays.

Shutdown (Unix) Poweroff

1. Exit all user terminals from Gold by pressing the F9 or F4 key, until the system prompts Do you really want to log off the system? NO
2. Enter Y for Yes, or press the space bar to change to YES and , when the terminal should display the login prompt.
3. From the Main Console, at the login enter root or , then enter your root password (abacus, apricot etc) and or .
4. At the root prompt, normally $ or #, enter down, which will bring the system down immediately.
5. The system will process the shutdown routines. Wait. The screen display will finally show the following message:

   Safe to power off

   Or

   Press any Key to Reboot
6. You may now switch off the power or press any key to restart the machine.
7. It is recommended to regularly carry out "power off" before rebooting – leaving the power off for at least 60 seconds.

4.18 SUMMARY

In this unit, we have learn about the basic concept of unix commads and some important basic and essential command.

4.19 CHECK YOUR PROGRESS

- **What do you mean by command?**

  ……
  ……
  ……
  ……

- **What are the Advantages and Disadvantages of UNIX operating system?**

  ……
  ……
  ……
  ……
  ……
  ……

- **Discuss the major file related commands**

  ……
  ……
  ……
  ……

- **Discuss directory related commands**

  ……
  ……
  ……
  ……
  ……
  ……

Further Reading:

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education

BLOCK-2 INTRODUCTION

This block is a course on "**Redirection, vi editor, processes in UNIX**". This block course contains three units. Unit 1 is **I/O redirection and piping** , With this unit, the learners will be acquainted with some important concepts like Standard streams: standard input, standard output, standard error, operator , piping, and commonly used filter: cat, pg, more, head, tail, grep, sort, nl, pr, wc, tee, uniq, tr, cut, paste, lpr etc

. Unit 2 discusses the Working with **Vi editor**. This includes the Introduction to vi editor,modes, status line commands, Opening & modifying a file, Saving a file and exiting vim, Search and Replace, undoing changes, yanking, Accessing multiple files, Window Commands, Interacting with system, Macros, vim configuration basics, syntax of ex commands, Addresses, Address symbols, options

. Unit 3 is the last unit and it discusses about **Processes in UNIX**.and cover the topic lke process fundamentals, connecting processes with pipes, Redirecting input output, manual help, Background processing, killing processes, managing multiple processes, changing process priority, scheduling of processes: at, batch, crontab command

The learners will learn how to work with **Vi editor**. The units of this block also discusses abut process in Unix Operating System and **I/O redirection and piping.** you will notice some boxes along-side, which have been included to help you know some of the difficult, unseen terms. Some "ACTIVITY' (s) have been included to help you apply your own thoughts. Again, we have included some relevant concepts in "LET US KNOW" along with the text. And, at the end of each section, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be better if you solve the given problems in these boxes immediately, after you finish reading the section in which these questions occur and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

UNIT-5 IO REDIRECTION AND PIPING

Structure:

5.1 INTRODUCTION

Linux/unix redirection features offer a powerful suite of tools for enhancing various workflows. Adhering to the "Unix philosophy," which emphasizes creating tools that excel at one specific task,

modern command-line tools have built on this principle. Each tool is highly effective on its own but becomes even more powerful when used in combination. Mastering the manipulation of I/O streams, whether you're developing complex software or simply working at the command line, can significantly boost your efficiency.

5.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of Standard streams in UNIX operating system
- Learning about piping
- commonly used filter commands

5.2 STREAMS

Input and output in the Linux/Unix environment is distributed across three *streams*. These streams are:

- **standard input (*stdin*)**
- **standard output (*stdout*)**
- **standard error (*stderr*)**

The streams are also numbered:

- **stdin** (**0**)
- **stdout** (**1**)
- **stderr** (**2**)

In typical interactions with a terminal, standard input is provided by the user's keyboard, while standard output and standard error are shown as text on the user's screen. Together, these three channels are known as the standard streams.

Standard Input

The standard input stream usually delivers data from the user to a program, with input typically coming from a device like a keyboard.

Standard Output

Standard output refers to the data produced by a program. By default, when the standard output stream isn’t redirected, it sends text directly to the terminal. You can test this by using the `echo` command to display some arbitrary text. For example:

```
Copy

$ echo Sent to the terminal
```

```
Output
Sent to the terminal
```

When used without any additional options, the `echo` command outputs any argument that is passed to it on the command line.

Run echo without any arguments:

It will return an empty line. Some programs do not do anything without provided arguments.

Standard Error

Standard error captures error messages produced by a program when it encounters problems. By default, this stream also outputs to the terminal.

To illustrate standard error, consider the `ls` command, which lists the contents of a directory. If you run `ls` without any arguments, it shows the contents of the current directory. However, if you provide a directory that doesn't exist or is inaccessible, `ls` will generate an error message that is sent to standard error. For example:

Since % is not an existing directory, this will send the following text to standard error:

```
Output
ls: cannot access %: No such file or directory
```

A program doesn't need to crash or complete to generate standard error; the choice of output stream is determined by the program's design. Standard output and standard error are technically similar, but they serve different purposes. Standard output is generally used for regular program output, while standard error is reserved for error messages. Some tools and scripts rely on the convention that an empty standard error stream signifies successful execution. Additionally, programs might use standard error to report minor issues without halting or affecting the primary output. The distinction between these streams helps separate expected results from error or diagnostic information.

Stream Redirection

Linux/Unix includes redirection commands for each stream. These can be used to write standard output or standard error to a file. If you write to a file that does not exist, a new file with that name will be created prior to writing.

Commands with a single bracket *overwrite* the destination's existing contents.

Overwrite

- > - standard output

- < - standard input
- **2>** - standard error

Commands with a double bracket *do not* overwrite the destination's existing contents.

Append

- >> - standard output
- << - standard input
- **2>>** - standard error

5.3 PIPES

Pipes allow you to redirect the output from one program and use it as input for another. When you use a pipe, the standard output of the first program is passed directly to the second program, rather than being displayed on the terminal. As a result, only the final output from the second program will be shown.

The Linux ***pipe*** is represented by a vertical bar: |

Here is an example of a command using a pipe:

```
$ ls | less
```

In this example, using a pipe with the ls command sends its output—showing the contents of your current directory—to the less program. The less program then displays this data one screen ful at a time, allowing you to scroll through the directory contents easily.

While ls typically shows the directory contents across multiple rows, piping it to less formats the output so that each entry appears on a new line and can be viewed page by page.

Pipes (|) differ from redirection operators (> and >>). Pipes redirect the output of one command to be used as input for another command, whereas > and >> are used to redirect output directly to files.

5.5 FILTERS

Filters are programs designed to process plain text—either from files or generated by other commands—by transforming it into a useful format. These programs read from standard input (stdin) and write their results to standard output (stdout). Linux/Unix provides a variety of filters, each serving different purposes. Here are some commonly used filters:

- **grep**: Searches for patterns within text. It outputs lines that match the specified pattern.
- **awk**: A powerful text processing tool that allows for pattern scanning and processing, often used for extracting and manipulating data from files or command outputs.
- **sed**: A stream editor for performing basic text transformations on an input stream (a file or input from a pipe).
- **sort**: Organizes lines of text files in a specified order (e.g., alphabetical or numerical).

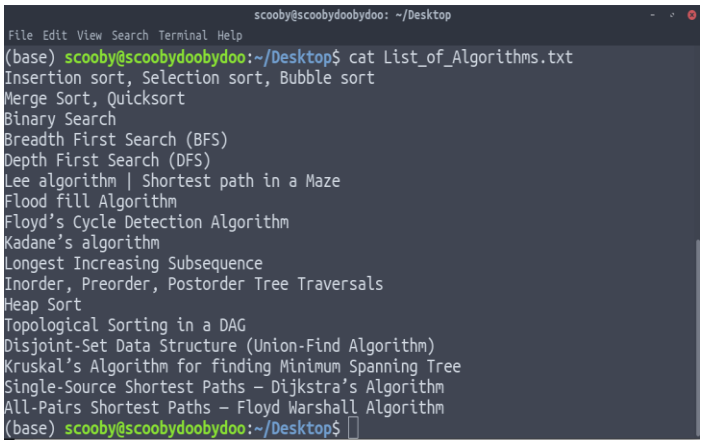
- **uniq**: Filters out repeated lines in a file or input, typically used in conjunction with sort.
- **cut**: Extracts sections from each line of files or input, based on specified delimiters or field positions.

5.6 SOME IMPORTANT BOTTOM OF FORMUNIX FILTER COMMANDS

1. **cat :** Displays the text of the file line by line.

   Syntax:

   cat [path]



2. **Pg command**

   Use the **pg** command to read the files named in the **File** parameter and writes them to standard output one screen at a time. If you specify hyphen (-) as the **File** parameter or run the **pg** command without options, the **pg** command reads standard input. Each screen is followed by a prompt. If you press the Enter key, another screen displays. Subcommands used with the **pg** command let you review content that has already passed.

3. **More command**

   more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large . The more command also allows the user do scroll up and down through the page. The syntax along with options and command is as follows. Another application of more is to use it with some other command after a pipe. When the output is large, we can use more command to see output one by one.

   Syntax:

   more [-options] [-num] [+/pattern] [+linenum] [file_name]

   - **[-options]**: any option that you want to use in order to change the way the file is displayed. Choose any one from the followings: (-d, -l, -f, -p, -c, -s, -u)
   - **[-num]**: type the number of lines that you want to display per screen.
   - **[+/pattern]**: replace the pattern with any string that you want to find in the text file.
   - **[+linenum]**: use the line number from where you want to start displaying the text content.
   - **[file_name]**: name of the file containing the text that you want to display on the screen.

While viewing the text file use these controls:

Enter key: to scroll down line by line.

Space bar: To go to the next page.

b key: To go to back one page.

Options:

- **-d :** Use this command in order to help the user to navigate. It displays "[Press space to continue, 'q' to quit.]" and displays "[Press 'h' for instructions.]" when wrong key is pressed.

  Example:

  more -d sample.txt

- **-f :** This option does not wrap the long lines and displays them as such.

  Example:

  more -f sample.txt

- **-p :** This option clears the screen and then displays the text.

  Example:

  more -p sample.txt

- **-c :** This command is used to display the pages on the same area by overlapping the previously displayed text.

  Example:

  more -c sample.txt

- **-s :** This option squeezes multiple blank lines into one single blank line.

  Example:

  more -s sample.txt

- **-u :** This option omits the underlines.

  Example:

  more -u sample.txt

- **+/pattern :** This option is used to search the string inside your text document. You can view all the instances by navigating through the result.

  Example:

  more +/reset sample.txt

- **+num :** This option displays the text after the specified number of lines of the document.

  Example:

  more +30 sample.txt

Using more to Read Long Outputs: We use more command after a pipe to see long outputs. For example, seeing log files, etc.

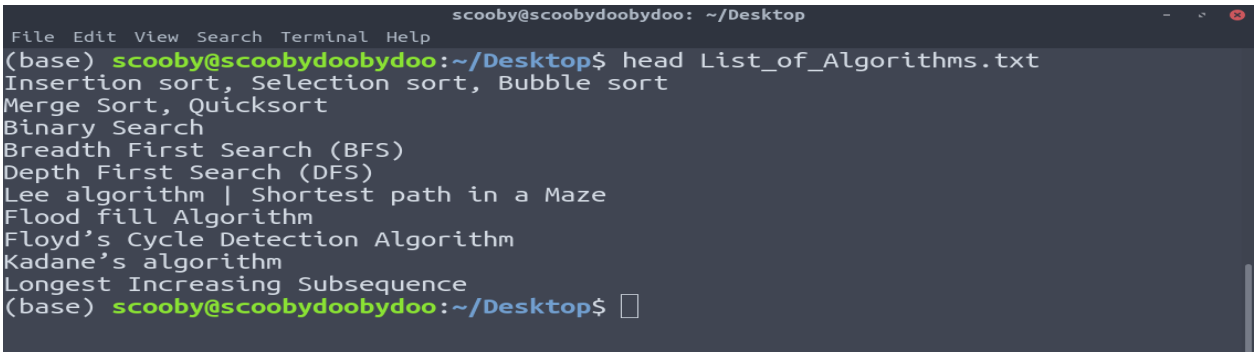
cat a.txt | more

4. **head** :

   Displays the first n lines of the specified text files. If the number of lines is not specified then by default prints first 10 lines.

   Syntax:

   head [-number_of_lines_to_print] [path]



5. **tail:** It works the same way as head, just in reverse order. The only difference in tail is, it returns the lines from bottom to up.

   Syntax:

   tail [-number_of_lines_to_print] [path]

6. **sort :** Sorts the lines alphabetically by default but there are many options available to modify the sorting mechanism. Be sure to check out the main page to see everything it can do.

   Syntax:

   sort [-options] [path]

7. **Uniq:** Removes duplicate lines. uniq has a limitation that it can only remove continuous duplicate lines(although this can be fixed by the use of piping). Assuming we have the following data.

   Syntax:

   uniq [options] [path]

   You can see that applying uniq doesn't remove any duplicate lines, because **uniq** only removes duplicate lines which are together.

   When applying uniq to sorted data, it removes the duplicate lines because, after sorting data, duplicate lines come together.

8. **wc_:** wc command gives the number of lines, words and characters in the data.

   Syntax:

   wc [-options] [path]

9. **grep:** grep is used to search a particular information from a text file.

Syntax:

grep [options] pattern [path]

Below are the two ways in which we can implement grep.

10. **tac :** tac is just the reverse of cat and it works the same way, i.e., instead of printing from lines 1 through n, it prints lines n through 1. It is just reverse of cat command.

Syntax:

tac [path]

11. **Sed:** sed stands for stream editor. It allows us to apply search and replace operation on our data effectively. *sed* is quite an advanced filter and all its options can be seen on its man page.

Syntax:

sed [path]

The expression we have used above is very basic and is of the form **'s/search/replace/g'**

In the above image, we can clearly see that Scooby is replaced by Scrapy.

12. **nl :** nl is used to number the lines of our text data.

Syntax:

nl [-options] [path]

13. **pr Command**

The **pr** command does minor formatting of files on the terminal screen or for a printer. For example, if you have a long list of names in a file, you can format it onscreen into two or more columns.

Following is the syntax for the **pr** command –

pr option(s) filename(s)

The **pr** changes the format of the file only on the screen or on the printed copy; it doesn't modify the original file. Following table lists some **pr** options −

Sr.No.	Option & Description
1	**-k** Produces **k** columns of output
2	**-d** Double-spaces the output (not on all **pr** versions)
3	**-h "header"** Takes the next item as a report header

4 **-t**

Eliminates the printing of header and the top/bottom margins

5 **-l PAGE_LENGTH**

Sets the page length to PAGE_LENGTH (66) lines. The default number of lines of text is 56

6 **-o MARGIN**

Offsets each line with MARGIN (zero) spaces

7 **-w PAGE_WIDTH**

Sets the page width to PAGE_WIDTH (72) characters for multiple text-column output only

Before using **pr**, here are the contents of a sample file named food.

```
$cat food
Sweet Tooth
Bangkok Wok
Mandalay
Afghani Cuisine
Isle of Java
Big Apple Deli
Sushi and Sashimi
Tio Pepe's Peppers
.....
$
```

Let's use the **pr** command to make a two-column report with the header *Restaurants* −

```
$pr -2 -h "Restaurants" food
Nov  7  9:58 1997  Restaurants   Page 1

Sweet Tooth            Isle of Java
Bangkok Wok            Big Apple Deli
Mandalay               Sushi and Sashimi
Afghani Cuisine        Tio Pepe's Peppers
.....
```

$

14. lp and lpr Commands

The command **lp** or **lpr** prints a file onto paper as opposed to the screen display. Once you are ready with formatting using the **pr** command, you can use any of these commands to print your file on the printer connected to your computer.

Your system administrator has probably set up a default printer at your site. To print a file named **food** on the default printer, use the **lp** or **lpr** command, as in the following example −

```
$lp food
request id is laserp-525  (1 file)
$
```

The **lp** command shows an ID that you can use to cancel the print job or check its status.

- If you are using the **lp** command, you can use the -n**Num** option to print Num number of copies. Along with the command **lpr**, you can use -**Num** for the same.
- If there are multiple printers connected with the shared network, then you can choose a printer using -d**printer** option along with lp command and for the same purpose you can use -P**printer** option along with lpr command. Here printer is the printer name.

15. The lpstat and lpq Commands

The **lpstat** command shows what's in the printer queue: request IDs, owners, file sizes, when the jobs were sent for printing, and the status of the requests.

Use **lpstat -o** if you want to see all output requests other than just your own. Requests are shown in the order they'll be printed −

```
$lpstat -o
laserp-573  john  128865  Nov 7  11:27  on laserp
laserp-574  grace  82744  Nov 7  11:28
laserp-575  john   23347  Nov 7  11:35
$
```

The **lpq** gives slightly different information than **lpstat -o** −

```
$lpq
laserp is ready and printing
Rank   Owner      Job  Files                 Total Size
active john       573  report.ps             128865 bytes
1st    grace      574  ch03.ps ch04.ps        82744 bytes
2nd    john       575  standard input         23347 bytes
$
```

Here the first line displays the printer status. If the printer is disabled or running out of paper, you may see different messages on this first line.

16. The cancel and lprm Commands

The **cancel** command terminates a printing request from the **lp command**. The **lprm** command terminates all **lpr requests**. You can specify either the ID of the request (displayed by lp or lpq) or the name of the printer.

$cancel laserp-575

request "laserp-575" cancelled

$

To cancel whatever request is currently printing, regardless of its ID, simply enter cancel and the printer name −

$cancel laserp

request "laserp-573" cancelled

$

The **lprm** command will cancel the active job if it belongs to you. Otherwise, you can give job numbers as arguments, or use a **dash (-)** to remove all of your jobs −

$lprm 575

dfA575diamond dequeued

cfA575diamond dequeued

$

The **lprm** command tells you the actual filenames removed from the printer queue.

17. tr command D

The **tr** command is a UNIX command-line utility for translating or deleting characters. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters, and basic find and replace. It can be used with UNIX pipes to support more complex translation. **tr stands for translate.**

Syntax :

$ tr [OPTION] SET1 [SET2]

Options -c : complements the set of characters in string.i.e., operations apply to characters not in the given set -d : delete characters in the first set from the output. -s : replaces repeated characters listed in the set1 with single occurrence -t : truncates set1.

18. tee command:

Reads the standard input and writes it to both the standard output and one or more files. The command is named after the T-splitter used in plumbing. It basically breaks the output of a program so that it can be both displayed and saved in a file. It does both the tasks simultaneously, copies the result into the specified files or variables .

SYNTAX:

tee [OPTION]... [FILE]...

Options :

1.-a Option :

It basically do not overwrite the file but append to the given file. Suppose we have

file1.txt

Input: uprtou
for
uprtou

and

file2.txt

Input: uprtou
for
uprtou

SYNTAX :

uprtou@HP:~$ wc -l file1.txt|tee -a file2.txt

OUTPUT :

3 file1.txt

uprtou@HP:~$cat file2.txt
OUTPUT:
uprtou
for
uprtou
3 file1.txt

19. cut command

The cut command in linux/unix is a command for cutting out the sections from each line of files and writing the result to standard output. It can be used to cut parts of a line by byte position, character, and field. The cut command slices a line and extracts the text. It is necessary to specify an option with a command otherwise it gives an error. If more than one file name is provided then data from each file is not preceded by its file name.

Syntax of cut Command

The basic syntax of the cut command is:

cut OPTION... [FILE]...

Where

`OPTION` specifies the desired behaviour

`FILE` represents the input file.

Note: If FILE is not specified, **`cut`** reads from standard input (stdin).

Options Available in cut Command

Here is a list of the most commonly used options with the **`cut`** command:

Option	Description
-b, –bytes=LIST	Selects only the bytes specified in LIST (e.g., -b 1-3,7).
-c, –characters=LIST	Selects only the characters specified in LIST (e.g., -c 1-3,7).
-d, –delimiter=DELIM	Uses DELIM as the field delimiter character instead of the tab character.
-f, –fields=LIS	Selects only the fields specified in LIST, separated by the delimiter character (default is tab).
-n	Do not split multi-byte characters (no effect unless -b or -c is specified).
–complement	Invert the selection of fields/characters. Print the fields/characters not selected.

20. Paste Command

Paste Command in Unix is among the most common and useful commands in Unix or Linux Shell Scripting. The main use of the paste command is to join the files horizontally by separating the lines from each file with a tab as a delimiter to get the required output. Paste command can be used to merge a single file or set of files aGenerally, the paste command is used both for single file handling and multiple file handling.

Syntax:

The basic syntax of the paste command in Unix Shell Scripting is given below:

```
paste [option].. [files]..
```

The main use of the paste command is to join the files horizontally by separating the lines from each file with a tab as a delimiter to get the required output.

Options that are available in paste commands are:

- **-d, delimiter:** Specifies the delimiter mentioned in the input parameter to display the output in a specified manner.
- **– – help:** This option helps to display the help message and options available in the paste command.
- **– – version:** Displays the version of the paste command that is running in your system and exit.

- **– s, -serial:** Helps display the file's contents in a horizontal format.

5.7 SUMMARY

In this unit we learn concept of **: IO redirection , piping** and filter.Standard streams: standard input, standard output, standard error, operator >, piping, commonly used filter: cat, pg, more, head, tail, grep, sort, nl, pr, wc, tee, uniq, tr, cut, paste, lpr, examples using streams and filters

5.8 CHECK YOUR PROGRESS

- **What do you mean by Stream?**

  ……
  ……
  ……
  ……
  ……

- **What is pipe?**

  ……
  ……
  ……
  ……
  ……

- **Discuss the major filter commands**

  ……
  ……
  ……
  ……
  ……

- **Discuss about tee command.**

  ……
  ……
  ……
  ……
  ……

5.9 FURTHER READING

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education
- https://www.redwood.com/article/unix-job-scheduling/
- https://www.tutorialspoint.com/unix/unix-basic-utilities.htm
- https://www.geeksforgeeks.org/tr-command-in-unix-linux-with-examples/

UNIT-6 VI EDITOR

Structure

6.1 INTRODUCTION

There are various methods to edit files in Unix, with the screen-oriented text editor vi being one of the most effective options. This editor allows you to modify lines of text in the context of the surrounding content. Additionally, an enhanced version of vi called VIM (Vi IMproved) is now available.

vi is widely recognized as a standard Unix editor due to several reasons:

- It is typically present on all Unix variants.
- Its implementations are quite consistent across different systems.
- It requires minimal system resources.

- It is more user-friendly compared to other editors like ed or ex.

You can use vi to edit an existing file, create a new file, or simply read a text file. VIM, which stands for Vi IMproved, builds upon this functionality.

vi employs a variety of internal commands for navigating through text files and editing their contents. It also facilitates copying and moving text both within the same file and across different files. The editor relies heavily on keyboard input, with nearly every key performing a specific function.

To create or edit a file with vi, you would use the following command:

If the specified file does not exist, vi will open a new file with a screen that displays the filename at the bottom with the label [New File]. The cursor will be positioned at the top of the screen, with all other lines except the last one showing a ~ to indicate nonexistent lines. The last line is used for command input and also displays system messages.

6.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of **vi editor** in UNIX operating system
- Modes of Vi editor
- Operations on vi Editor
- status line commands,
- how to Opening & modifying a file, Saving a file
- Search and Replace, undoing changes,
- Accessing multiple files,
- basics, syntax of ex commands,
- options

6.2 MODES

The vi editor has three main modes:

- **Command Mode:** When you first open a file with the vi editor, you are in command mode. In this mode, you can move the cursor around the file and issue commands to the editor. The most important commands in this mode are the movement commands, which allow you to move the cursor around the file. These commands are very important because they allow you to edit the file in a very precise way.
- **Insert Mode:** Itis the mode you are in when you want to insert or change the text in the file. In this mode, the cursor is positioned at the point where you want to make the change, and you can type in the new text. After making the change, you can press the Esc key to return to command mode.

- **View Mode:** When you want to view the file but not make any changes to it, that's the view mode of the vi editor. This mode is useful for viewing files you do not want to modify, such as configuration files. In view mode, you can move the cursor around the file and view the contents of the file. However, you cannot change the file in this mode. You can press the Esc key to exit view mode and return to command mode.

6.3 OPERATIONS ON VI EDITOR

1. vi Command Mode:

   The "vi" command mode is the default mode for the vi editor. In this mode, characters typed on the keyboard are interpreted as commands. The commands available in this mode allow you to move the cursor around the screen, search for text, delete text, change text, and so on.

   There are two ways to enter the "vi" command mode:

   - Use the Esc key
   - Type the ":" (colon) character

2. vi Editor Insert mode:

   You must first be in insert mode to insert text while in the vi editor. To enter insert mode, press the "i" key. You can also enter insert mode by pressing the "a" key, which will insert text after your cursor's character. You can also enter insert mode by pressing the "I" key, which will insert text at the beginning of the current line. Finally, you can enter insert mode by pressing the "A" key, which will insert text at the end of the current line.

   Once you are in insert mode, you can start typing to insert text. To exit insert mode and return to command mode, press the "Esc" key.

3. How to Use vi Editor:

   - For command mode: Press the Esc key
   - For insert mode: Press the "i" key
   - For last-line mode: Press the ":" key

   In command mode, you can issue commands to the vi editor. For example, you can use the h, j, k, and l keys to move the cursor around the file. You can use the w, b, and e keys to move the cursor forward or backward by word or by sentence. You can use the 0 (zero) key to move the cursor to the beginning of the current line and the $ key to move the cursor to the end of the current line. Press i to enter the insert mode. In this mode, you can insert text into the file. The text is inserted at the location of the cursor.

   In last-line mode, you can enter commands that affect how the vi editor behaves. For example, you can use the: set command to turn on or off certain features of the vi editor. To enter last-line mode, press the: key. To exit the last-line mode, press the Esc key.

4. vi Editing Commands:

   There are a variety of commands for editing text in vi Editor. These commands can be used to move the cursor around, delete text, insert text, and so on. Each command is typically a single letter. For example, the "i" command inserts text at the current cursor.

5. Moving Within a File:

   Moving within a file means moving the cursor to a new location in the same file. The cursor is moved using the arrow keys or other keys. The other keys include the following:

   - h – move left
   - j – move down
   - k – move up
   - l – move right

   The moving can also be done using keyboard shortcuts. For example, the Ctrl + F key combination moves the cursor forward one full screen, and the Ctrl + B key combination moves the cursor backward one full screen.

6. Saving and Closing the File:

   In the vi editor, to save the file without exiting the editor, we can use ":wq." This saves and exits the file. Otherwise, to save the file, we use:w. To exit the file without saving, we use ":q."

6.4 ADVANTAGES OF VI EDITOR

There are various advantages of using the vi editor, which include:

1. vi Editor is Powerful

   vi is a very versatile text editor that can be used to edit various files. It is used to edit files in many different programming languages, including C, C++, Java, and Python. "vi" can also be used to edit HTML, XML, and CSS files. It is a very user-friendly text editor with a wide variety of features that make it a great choice for editing text files. "vi" is a very popular text editor used by many people, including programmers, web developers, and system administrators.

2. Provides a Wide Range of Features

   "vi" editor is a powerful text editor that comes with a wide range of features that can be used to edit text files. Some of the most notable features include:

   - **Support for a Wide Range of File Formats:** vi editors can be used to edit files in various formats, including plain text, HTML, XML, and more.
   - **Syntax Highlighting:** vi editors can highlight the syntax of various programming languages, making it easier to read and edit code.
   - **Code Completion:** vi editors can automatically complete code for you based on your editing context. This can save a lot of time, especially when working with large codebases.
   - **Macro Recording:** vi editors can record your keystrokes and play them back, which can be very useful for repetitive tasks.
   - **Plug-in Support:** vi editors can be extended with plug-ins, adding new features or customizing the editor to suit your needs.

3. vi Editor Is Easy to Use and Provides a User-Friendly Interface

   Advantages of the vi editor include ease of use and user-friendly UI. It is a text editor that programmers and system administrators widely use.

The **vi editor** is easy to learn and use and provides various features that make it a powerful tool for editing text files. It is available on most Unix-like operating systems and is the default editor on many Linux distributions.

4. vi Editor Is Highly Configurable and Customizable

   The vi editor is highly configurable and can be customized to suit the user's needs. This editor can be customized to work with any programming language, and it has many features that make it an ideal choice for developers who want to work with code.

5. vi Editor is Free and Can Be Downloaded From the Internet

   The **vi editor** is a text editor available for free and can be downloaded from the internet. This editor has many features that make it a powerful text editor, such as syntax highlighting, code completion. It is also available for many different operating systems, making it a versatile tool for many different users.

6. vi Editor Is a Cross-Platform Text Editor

   The vi editor is a cross-platform text editor that can be used on various operating systems. This is an advantage because the vi editor will work on various devices, including PCs, Macs, and Linux computers. It is also available for mobile devices, such as smartphones and tablets. You can use vi to edit text on the go, which is convenient if you need to make changes to a document while you are away from your computer.

6.5 STATUS LINE COMMANDS

vi contains a huge array of commands, many of which have overlapping functions. At first, it's quite normal for new users to feel overloaded by this. The purpose of this heading , however, is to provide you with an overview of the most essential vi commands. As you begin to use vi, you'll find that it is an extremely powerful text editor, and that it may take you a while to become proficient.

Note that there is a read-only version of vi called view. When you open a file with view, you can use vi commands, but you cannot write (or save) your changes. This allows you or others to read a vi file without accidentally changing it.

Starting vi

In the sub-sections that follow, you'll learn how to start vi, enter text in a file, save (write) the file, and quit vi.

Creating a File

Start vi and edit the file paint as shown in this example:

$ vi paint

If paint already exists, vi will open the existing file; if this is a new file, vi will create it. For the purposes of this example, paint should be a new file.

The vi editing screen appears in a moment:

Figure The vi Editing Screen

The cursor appears in the upper left corner of the screen. Blank lines are indicated by a vertical series of tildes (~).

Note that you can also start vi without specifying a file name by just typing vi. You can then name the file later when you exit vi.

The Status Line

The last line of the screen, called the status line, shows the name of the file and the number of lines and characters in the file. When you create a new file, as is the case with our example, the status line indicates that it's a new file.

6.6 THE TWO MODES OF VI

There are two modes of operation in vi: entry mode and command mode. You use entry mode to enter text into a file, while command mode is used to enter commands that perform specific vi functions. Command mode is the default mode for vi.

Since vi doesn't indicate which mode you're currently in, distinguishing between command mode and entry mode is probably the single greatest cause of confusion among new vi users. However, if you remember just a few basic concepts from the beginning, you should be able to avoid most of the usual "vi stress."

When you first open a vi file, it's always in command mode. Before you can enter text in the file, you must type one of the vi entry commands, such as i ("insert"), to insert text **at** the current cursor location, or a ("append"), to insert text **after** the current cursor location. (These and other vi entry commands are covered in greater detail later in this chapter.)

Whenever you want to return vi to command mode, press Esc. If you're not sure which mode vi is presently in, simply press Esc to make sure it's in command mode and continue from there. If you press Esc while vi is already in command mode, the system beeps and the screen flashes, but no harm is done.

1. Entry Mode

   To enter text in the sample file paint, type the vi "insert" command i. This takes vi out of command mode and puts it into entry mode.

   Now type a few short lines of text, ending every line with a Return. Characters you type appear to the left of the cursor and push any existing characters to the right. For the moment, you can

correct your mistakes by backspacing and retyping a line before you press Return. Later you will learn how to edit the text you entered.

When you finish entering text in paint, press Esc to return to command mode. The cursor moves back onto the last character entered. Now you can enter more vi commands.

If vi seems to act unpredictably, make sure that you are not in "Caps Lock" mode, which would cause your entries to be all capital letters. On some systems, the F1 key (which is usually located next to the Esc key) acts as the Caps Lock. Pressing this key instead of Esc is a common error.

2. Command Mode

When you open a file with vi, you are in command mode. In this mode, you can enter commands to implement a wide range of functions. Most vi commands consist of one or two letters and an optional number. Usually, there are upper and lowercase versions of commands that perform related but different functions. As an example, typing a appends the file to the right of the cursor, while typing A appends the file at the **end** of the line.

Most vi commands don't require that you press Return to execute them. Commands beginning with a colon (:), however, do require that you press Return after the command. Some discussions of the vi editor refer to commands preceded with a colon as a third, and uniquely separate mode of vi, **last-line mode**. This is because when you type the colon while in command mode, the colon and the remainder of what is typed appear on the bottom line of the screen. For the purpose of this discussion, however, all vi commands are initiated from command mode.

Commands preceded with a colon are actually **ex** commands. vi and ex are two separate interfaces to the same text editing program. While vi is a screen-oriented interface, ex is a line-oriented interface. The full set of ex commands is available from within vi. When you press the colon, you are actually switching to the line-oriented, ex interface. This allows you to perform many file manipulation commands without ever leaving vi.

6.7 ENDING A SESSION

When you edit a file in vi, your changes are not made directly to the file. Instead, they are applied to a copy of the file that vi creates in a temporary memory space called the **buffer**. The permanent disk copy of the file is modified only when you **write** (save) the contents of the buffer.

This arrangement has its good and bad points. On the one hand, it means that you can quit a file and discard all the changes that you have made during an editing session, leaving the disk copy intact. On the other hand, you could lose the (unsaved) contents of the work buffer if the system crashes. (People on remote terminals connected by phone lines are especially vulnerable to unplanned interruptions.)

The best policy is to save your work frequently, especially when making substantive changes.

6.8 SAVING CHANGES AND QUITTING VI

vi is rich in more or less synonymous commands that control saving the buffer contents to a file and quitting vi. These commands give you the option of saving, saving-and-quitting, or quitting-without-saving.

Saving

Save the contents of the buffer (write the buffer to the file on disk) by typing:

:w

followed by Return.

Saving and Quitting

Save and quit by typing:

:wq

followed by Return. Alternatively, type ZZ.

Note that the command ZZ is neither preceded by a colon nor followed by Return.

Quitting Without Saving

When you've made no changes to a file and simply want to quit, type:

:q

followed by Return.

Printing a File

Once you have quit a vi file, you can print the file with the following command:

```
$ lp filename
```

where **filename** is the name of the vi file to be printed. This command prints the file to your default printer. The file is printed without any formatting, line for line, just as it appears on the screen.

6.9 BASIC VI COMMANDS

The following sections explain several categories of vi commands. These include:

- Moving around in a file
- Inserting text
- Changing and substituting text
- Undoing changes to text
- Deleting text
- Copying and moving text
- Repeating commands

1. Moving Around in a File

   In the previous sections you learned how to create, save, print, and exit a vi file. Now that you have created a file, you'll need to understand the concepts required to navigate within it. Open your practice file now, and try out each of the commands discussed in this section

   Moving the Cursor

   When you start vi, the cursor is in the upper left corner of the vi screen. In command mode, you can move the cursor with a number of keyboard commands. Certain letter keys, the arrow keys, and the Return key, Back Space (or Delete) key, and the Space Bar can all be used to move the cursor when you're in command mode.

Note -

Most vi commands are case-sensitive; the "same" command typed in lowercase and uppercase characters could have radically different effects.

Moving with Arrow Keys

If your machine is equipped with arrow keys, try these now. You should be able to move the cursor freely about the screen using combinations of the up, down, right, and left arrow keys. Notice that you can only move the cursor across already existing text or input spaces.

If you're using vi from a remote terminal, the arrow keys may not work correctly. This will depend on your terminal emulator. If the arrow keys don't work in your case, you can use the following substitutes:

- To move left, press h.
- To move right, press l.
- To move down, press j.
- To move up, press k.

1.Moving One Word

Press w ("word") to move the cursor to the right one word at a time.

Press b ("back") to move the cursor to the left one word at a time.

Press W or B to move the cursor past the adjacent punctuation to the next or previous blank space.

Press e ("end") to move the cursor to the last character of the current word.

Moving to Start or End of Line

Press ^ to move the cursor to the start of the current line.

Press $ to move the cursor to the end of the current line.

Moving Down One Line

Press the Return key to move the cursor to the beginning of the next line down.

Moving Left

Press the Back Space key to move the cursor one character to the left.

Moving Right

Press the Space Bar to move the cursor one character to the right.

Moving to the Top

Press H ("high") to move the cursor to the top of the screen.

Moving to the Middle

Press M ("middle") to move the cursor to the middle of the screen.

Moving to the Bottom

Press L ("low") to move the cursor to the bottom of the screen.

Paging and Scrolling

If you move down when the cursor is at the bottom of the screen, or move up when the cursor is at the top of the screen, you will see the text scroll up or down. This can be an effective way to display more text in a very short file, but it can be tedious to move this way through a long file.

You may have noticed that moving the cursor either past the bottom or past the top of the screen has the effect of scrolling text up or down. This works for a very short file, but it is a tedious way to move through a long file.

You can page or scroll backward or forward through a file, a screen or a half-screen at a time. (To try out these commands on paint, you might want to add text so you have a longer file to work with.)

Note that there is a fundamental difference between paging and scrolling. Scrolling actually scrolls the cursor up or down through the text **a line at a time,** as though it were on a paper scroll. Paging moves the cursor up or down through the text **a screen full at a time**. On a fast system, you might not notice the difference. However, if you're working from a remote terminal or in some other situation where your system is running slower than usual, this difference can become painfully apparent.

Page Forward One Screen

To scroll forward (move down) one screenful, pressCtrl-F. (Hold down the Control key and press the F key.) The cursor moves to the upper left corner of the new screen.

Scroll Forward One-Half Screen

To scroll forward one half of a screen, press Ctrl-D.

Page Backward One Screen

To scroll backward (i.e., move up) one screenful, press Ctrl-B.

Scroll Backward One-Half Screen

scroll backward one half of a screen, press Ctrl-U.

2. Inserting Text

vi provides many commands for inserting text. This section introduces you to the most useful of these commands. Note that each of these commands places vi in entry mode. To use any of these commands, you must first be in command mode. Remember to press Esc to make sure you are in command mode.

Append

Type a (append) to insert text to the **right** of the cursor. Experiment by moving the cursor anywhere on a line and typing a, followed by the text you want to add. Press Esc when you're finished.

Type A to add text to the **end** of a line. To see how this works, position the cursor anywhere on a text line and type A. The cursor will move to the end of the line, where you can type your additions. Press Esc when you're done.

Insert

Insert text to the left of the cursor by typing i from command mode.

Type I to insert text at the beginning of a line. (The command will move the cursor from any position on that line.) Again, as with all the commands in this section, press Esc to return to command mode after entering the desired text.

Open Line

Use these commands to open new lines, either above or below the current cursor position.

Type o to open a line **below** the current cursor position. To experiment, type o followed by a bit of text. You can enter several lines of text if you like. Press Esc when you are finished.

Type O to open a line **above** the current cursor position.

3. Changing Text

Changing text involves substituting one section of text for another. vi has several ways to do this, depending on circumstances.

Changing a Word

To replace a word, position the cursor at the beginning of the word to be replaced. Type cw, followed by the new word. To finish, press Esc.

To change **part** of a word, place the cursor on the word, to the **right** of the portion to be saved. Type cw, enter the correction, and press Esc.

Changing a Line

To replace a line, position the cursor anywhere on the line and type cc. The line disappears, leaving a blank line for your new text (which can be of any length). Press Esc to finish.

Changing Part of a Line

To replace part of a line, place the cursor to the **right** of the portion to be saved. Type C, enter the correction, and press Esc. This changes the portion of the line from the current cursor position to the end of the line.

Substituting Character(s)

To substitute one or more characters for the character under the cursor, type s, followed by the new text. Press Esc to return to command mode.

Replacing One Character

Use this command to replace the character highlighted by the cursor with another character. Position the cursor over the character and type r, followed by just one replacement character. After the substitution, vi automatically returns to command mode (there's no need to press Esc).

Transposing Characters

Correcting transposed characters takes just two keystrokes in vi. Suppose you find that you've typed "teh" when you meant to enter "the". Make the correction by putting the cursor over the first letter to be moved (in this case, e), and then type xp. The e and h will trade places - and vi will automatically return to command mode.

Breaking or Joining Lines

To break a line without affecting text, move the cursor to a space where you want the line to break and type r (for "replace") followed by Return. Note that if you type r with the cursor on a character and then press Return, that character will be replaced by the Return.

To join two lines, place the cursor on the upper line and type an uppercase J. (There's no need to press Esc after typing J.)

4. Undoing Changes

When editing text and making changes to a vi file, there will no doubt be times when you'll wish that you had **not** changed something. vi's undo commands allow you to back up one operation and continue on from there.

Undoing the Previous Command

If you make a mistake in vi or if you just change your mind once an operation is completed, you can undo your last command by pressing u immediately after the command. (There's no need to press Esc after typing u.) Pressing u a **second** time undoes the undo.

Undoing Changes to a Line

Type U to undo all changes you've made to a line. This command works only if you haven't moved the cursor off the line. (There's no need to press Esc after typing U.)

5. Deleting Text

These vi commands delete the character, word, or line you indicate. vi stays in command mode, so any subsequent text insertions must be preceded by additional commands to enter entry mode.

Deleting One Character

To delete one character, position the cursor over the character to be deleted and type x.

The x command also deletes the space the character occupied--when a letter is removed from the middle of a word, the remaining letters will close up, leaving no gap. You can also delete blank spaces in a line with the x command.

To delete one character before (to the left of) the cursor, type X (uppercase).

Deleting a Word or Part of a Word

To delete a word, position the cursor at the beginning of the word and type dw. The word and the space it occupied are removed.

To delete part of a word, position the cursor on the word to the **right** of the part to be saved. Type dw to delete the rest of the word.

Deleting a Line

To delete a line, position the cursor anywhere on the line and type dd. The line and the space it occupied are removed.

Deleting Part of a Line

You can also delete part of a line.

To delete everything to the **right** of the cursor, position the cursor to the right of the part of the line you want to save, and type D.

To delete everything to the **left** of the cursor, position the cursor to the right of the part of the line you want to delete and type d0 (d-zero).

Deleting to the End of the File

To delete everything from the current line to the end of the file, type dG. This also deletes the line on which the cursor is located.

Deleting from Beginning of File

To delete everything from the beginning of the file to the current line, type d1G. This also deletes the line on which the cursor is located.

6. Copying and Moving Text -- Yank, Delete, and Put

Many word-processors allow you to "copy and paste" and "cut and paste" lines of text. The vi editor also includes these features. The vi command-mode equivalent of "copy and paste" is **yank and put**; the equivalent of "cut and paste" is **delete and put**.

The methods for copying or moving small blocks of text in vi involves using a combination of the yank, delete, and put commands.

Copying Lines

To copy a line requires two commands: yy or Y ("yank") and either p ("put below") or P ("put above"). Note that Y does the same thing as yy.

To yank one line, position the cursor anywhere on the line and type yy. Now move the cursor to the line above where you want the yanked line to be put (copied), and type p. A copy of the yanked line will appear in a new line **below** the cursor.

To place the yanked line in a new line **above** the cursor, type P.

The yy command works well with a count: to yank 11 lines, for example, just type 11yy. Eleven lines, counting down from the cursor, will be yanked, and vi indicates this with a message at the bottom of the screen: 11 lines yanked.

You can also use the P or p commands immediately after any of the deletion commands discussed earlier. This puts the text you deleted above or below the cursor, respectively.

Moving Lines

Moving lines also requires two commands: dd ("delete") and either p or P.

To move one line, position the cursor anywhere on the line and type dd. For example, to delete 5 lines, type 5dd.

Next, move the cursor to the line above where you want the deleted line reinserted and type p. This inserts the text on a new line below the cursor.

Alternatively, you can put the deleted line above the cursor by typing P.

Using Named Buffers

To repeatedly insert a group of lines in various places within a document, you can yank (or delete) the lines into a named buffer. You specify named buffers by preceding a command with double quotes (") and a name for the buffer. For example, to yank four lines into the named buffer **a**, type "a4yy". You can use several different buffers. For example, you might also delete text from one location and add it to several others. To delete 12 lines into the named buffer **b**, type "b12dd".

To insert the text, precede the p or P command with *n*, where **n** is the named buffer. For example, to insert the lines saved in buffer **b**, type "bP".

You can overwrite named buffers with new lines. The buffers are saved until you exit vi.

When you use named buffers, you can safely delete and yank other text without affecting the lines you have already saved in the named buffers -- unless, of course, you purposely overwrite the named buffer.

6.10 EX COMMANDS

ex commands are more accurate and convenient than yank, delete, and put when you're dealing with large blocks of text. Rather than counting lines on the screen and then searching for an insertion point, you give vi a range of lines to be moved or copied and then specify the line before the insertion point. (Of course, with a delete command there is no insertion point.)

6.10.1 Turning Line Numbers On and Off

To turn line numbers **on**, type :set nu and press Return.

Line numbers appear in the left margin. Note that these numbers do not appear when you print out the file. They are visible only on the screen.

```
1 Oh, when I die, take my saddle from the wall,
2 Put it on my pony, lead him out of the stall.
3 Tie my bones to his back, point our faces to the west,
4 And we'll ride the prairies that we love the best.
5
6 Ride around, little doggies,
7 Ride around real slow.
8 Firey and Snuffy are rarin' to go.
~
~
~
~
~
~
:set nu
```

To turn line numbers **off**, type :set nonu and press Return.

6.10.2 Copying Lines

The basic form of the ex copy command is:

```
:line#,line# co line#
```

The first two numbers (separated by a comma) specify the range of lines to be copied. The third number is the line **before** the insertion point.

For example, to copy lines 1 through 5 of paint and place the copy after line 12, you would type the following:

```
:1,5 co 12
```

and press Return.

When specifying line ranges, use the abbreviations

- Period (.) to denote "from the current line."
- Dollar sign ($) to denote "to end of file."

Thus, to copy the range "from the current line through line 5" and insert this block after line 12, you would type:

```
:.,5 co 12
```

To copy the range "from line 6 through the end of the file" and insert this block after line 2, you would type:

```
:6,$ co 2
```

6.10.3 Moving Lines

The basic form of the ex move command is similar to the copy command discussed above:

```
:line#,line# m line#
```

Line ranges and insertion points are specified in the same ways, including use of the abbreviations . and $. The difference in function is simply that "move" removes a block from one location and reinserts it elsewhere.

For example, to move lines 1 through 5 to the line following 12, you would type:

```
:1,5 m 12
```

and press Return.

6.10.4 Deleting Lines

To delete a range of lines, use the command form:

```
:line#,line# d
```

For example, to delete lines 1 through 5, you would type:

```
:1,5 d
```

6.11 SEARCHING AND REPLACING WITH VI

vi provides several ways to find your place in a file by locating a specified string of characters. It also has a powerful global replacement function.

Finding a Character String

A **character string** is simply one or more characters in a row. It may include letters, numbers, punctuation, special characters, blank spaces, tabs, or carriage returns. A string may be a grammatical word or it may be part of a word.

To find a character string, type / followed by the string you want to search for, and then press Return. vi positions the cursor at the next occurrence of the string. For example, to find the string "meta", type /meta followed by Return.

Type n to go to the **next** occurrence of the string; type N to go to the **previous** occurrence.

To search backward in a file, you can use ? instead of /. In this case, the directions of n and N are reversed.

Searches normally are case-sensitive: a search for "china" will not find "China." If you want vi to ignore case during a search, type :set ic. To change it back to the default, case-sensitive mode, type :set noic.

If vi finds the requested string, the cursor will stop at its first occurrence. If the string is not found, vi will display Pattern not found on the last line of the screen.

Certain special characters (/ & ! . ^ * $ \ ?) have special significance to the search process and must be "escaped" when used in a search. To escape a special character, precede it with a backslash (\). For example, to search for the string "anything?" type /anything\? and press Return.

These special characters can be used as commands to the search function, so if you want to search for a string including one or more of these characters, you must indicate this by preceding the character with a backslash. To escape a backslash itself, type \\.

Refining the Search

You can make searches more precise by tagging the string with indicators for the following characteristics:

- Beginning of line
- End of line
- Beginning of word
- End of word
- Wild card characters

To match the beginning of a line, start the search string with a caret (^). For example, to find the next line beginning with "Search", type:

```
/^Search
```

To match the end of a line, end the search string with a dollar sign ($). For example, to find the next line ending with "search.", type:

```
/search\.$
```

(Note that the period is escaped with a backslash.)

To match the beginning of a word, type \< at the beginning of the string; to match the end of a word, type \> at the end of the string. Thus, to match a word, rather than a string, combine the end-of-word and beginning-of-word tags in the search pattern. For example, to find the next occurrence of the word--as opposed to the string--"search", type:

```
/\<search\>
```

To match any character, type a period (.) in the string at the location to be matched. For example, to find the next occurrence of "disinformation" or "misinformation", type:

```
/.isinformation
```

Since this is a search for a string and not a word, this search pattern may also find such constructions as "misinformationalist" and "disinformationism".

To search for alternative characters in a string, enclose the alternatives in brackets. The search pattern /[md]*string* will find strings beginning with either m or d. On the other hand, /[d-m]*string* will find strings beginning with any letter from d through m.

To match zero or more occurrences of the last character, type an asterisk (*) in the string. You can effectively combine brackets and the asterisk to look for well-defined alternatives. For example, to find all strings beginning with a through z and ending with "isinformation" **and** to find all occurrences of the string "isinformation", type:

```
/[a-z]*isinformation
```

Replacing a Character String

The procedure for replacing a text string is based on the search procedures discussed above. All the special matching characters for searches can be used in search-and-replace.

The basic command form is:

```
:g/search-string/s//replace-string/g
```

followed by the Return key.

Therefore, to replace every occurrence of the string "disinformation" with "newspeak", you would type:

```
:g/disinformation/s//newspeak/g
```

and press Return.

You can modify this command to halt the search and make vi query whether you want to make the replacement in each instance. The following command uses gc (adding c for "consult") to make vi stop at every occurrence of "disinformation" and ask whether you want to make the substitution. Respond with y for yes or n for no.

```
:g/disinformation/s//newspeak/gc
```

Going to a Specific Line

To go to the last line of an open file, by type G. To return to the first line of the file, by type 1G.

You can go to any other line by typing its number followed by G.

For example, suppose that you quit the file paint while editing line 51. You can access that line by opening the file and typing 51G.

Inserting One File into Another

vi makes it convenient to "read" (insert) a file into the file you are editing. The general form of the command is:

```
:line# r filename
```

If you do not specify a line number, vi inserts the file at the current cursor position.

For example, if you wanted to insert the file orwell at line 84 of the file paint, you would type:

```
:84 r orwell
```

Or you could position the cursor on line 84 and type:

```
:r orwell
```

6.12 EDITING MULTIPLE FILES

vi allows you to edit multiple files. For example, to edit the file or well while you are editing paint:

1. First, save your current work in paint. Type :w and press Return.
2. To edit orwell, type :n orwell and press Return.

3. Make editing changes to orwell and save your work.
4. When you are finished working with orwell and have saved your work, you have three choices:
    - Exit vi. Type :q and press Return.
    - Return to paint. Type :n # and press Return.
    - Swap back and forth between two files with the command :n #.

Editing a Series of Files

To edit a series of files, list the file names after the vi command when you start vi from the command prompt:

```
$ vi paint orwell
```

The files appear in the order in which they are listed. First paint appears. When you finish editing paint, type :n to go on to the next file, orwell. To go on to the next file without saving changes in the current file, type :n! instead of :n.

If you have a series of files with related names (for example, test1, test2, test3), you can use wildcard characters to specify a group of files:

```
$ vi test*
```

The files will appear for editing in alphabetical order by name.

Copying Lines Between Files

To copy lines from one file to another, do the following:

1. Edit the first file.
2. Save the desired lines in named buffers, using the yank command. For example, to save 10 lines in buffer **a**, type a10Y.
3. Without exiting vi, edit the next file (orwell in this example):

    ```
    :n orwell
    ```

4. Add the lines from the first file with the put command. For example, to put the contents of buffer **a** below the current cursor position, type ap.

Remember that the contents of all named buffers are lost whenever you exit vi. Don't use the quit (:q) command until you have finished any operations involving named buffers.

6.13 SETTING VI PARAMETERS

vi has many variables that affect its behavior and appearance. You can view a list of these variables (with their current settings) while running vi by typing:

```
:set all
```

followed by Return.

6.14 RECOVERING FROM A CRASH

If the system crashes, the contents of your buffer are at risk. Often, though, you can recover most of your work by restarting vi with the command form:

```
vi -r filename
```

where **filename** is the file you were editing at the time of the crash. The system will usually send you mail after the system is restarted, telling you there is a recover file.

6.15 HANDLING MULTIPLE FILES

VI uses last line mode to handle multiple files and buffers. A user can open as many buffers as required and switch from one to other. The commands are as below for multiple file handling.

Command & Description

: r fname : it reads(/insert) file fname below the current line.

:r !cmd : it reads(/insert) output of 'cmd' command below the current line.

:e fame: it stop editing current file, and edits the file fname.

:e !fname: it is similar to :e fname but after discarding chsnges made to current file.

:e! : it loads the last saved edition of current file.

Ctrl-^ or :e# : it returned most recently edited file.

:n : it edits next file.

:n! : it permits editing of next file without saving the current file.

:rew : it rewinds the file list to start editing first file.

:rew! : it permits editing to the 1st file in command line without saving the current file.

:f : it displays name of the current file.

:args : it displays name of all files in the buffer, the name of the current file is enclosed within square brackets.

:sp : it splits the existing window into 2 separate windows.

Ctrl-w/W : it cycles through window.

6.16 SUMMARY OF BASIC VI COMMANDS

The following table provides a convenient reference for basic vi commands.

Table 6-1 Basic vi Commands

Command	**Meaning**
Starting vi	
vi **filename**	Open or create file
vi	Open new file to be named later
vi **-r filename**	Recover crashed file
view **filename**	Open file read-only
Cursor Commands	
h	Move left one character
j	Move down one line
k	Move up one line
l	Move right one character
w	Move right one word
W	Move right one word (past punctuation)
b	Move left one word
B	Move left one word (past punctuation)

Command	Meaning
e	Move to end of current word
Return	Move down one line
Back Space	Move left one character
Space Bar	Move right one character
H	Move to top of screen
M	Move to middle of screen
L	Move to bottom of screen
Ctrl-F	Page forward one screen
Ctrl-D	Scroll forward one-half screen
Ctrl-B	Page backward one screen
Ctrl-U	Scroll backward one-half screen
Inserting Characters and Lines	
a	Insert characters to right of cursor
A	Insert characters at end of line
i	Insert characters to left of cursor

Command	Meaning
I	Insert characters at beginning of line
o	Insert line below cursor
O	Insert line above cursor
Changing Text	
cw	Change word (or part of word) to right of cursor
cc	Change line
C	Change from cursor to end of line
s	Substitute string for character(s) from cursor forward
r	Replace character at cursor with one other character
r Return	Break line
J	Join current line and line below
xp	Transpose character at cursor and character to right
~	Change case of letter (upper or lower)

Command	Meaning
u	Undo previous command
U	Undo all changes to current line
:u	Undo previous last-line command
Deleting Text	
x	Delete character at the cursor
X	Delete character to the left of the cursor
dw	Delete word (or part of word to right of cursor)
dd	Delete line containing the cursor
D	Delete part of line to right of cursor
dG	Delete to end of file
d1G	Delete from beginning of file to cursor
:5,10 d	Delete lines 5-10
Copying and Moving Text	
yy	Yank or copy line

Command	Meaning
Y	Yank or copy line
p	Put yanked or deleted line below current line
P	Put yanked or deleted line above current line
:1,2 co 3	Copy lines 1-2 and put after line 3
:4,5 m 6	Move lines 4-5 and put after line 6
Setting Line Numbers	
:set nu	Show line numbers
:set nonu	Hide line numbers
	Setting Case-sensitivity
:set ic	Searches should ignore case
:set noic	Searches should be case-sensitive
Finding a Line	
G	Go to last line of file
1G	Go to first line of file

Command	**Meaning**
21G	Go to line 21
Searching and Replacing	
/**string**	Search for **string**
?**string**	Search backward for **string**
n	Find next occurrence of **string** in search direction
N	Find previous occurrence of **string** in search direction
:g/*search*/s//*replace*/g	Search and replace
Clearing the Screen	
Ctrl-L	Clear (refresh) scrambled screen
Inserting a File into a File	
:r **filename**	Insert (read) file after cursor
:34 r **filename**	Insert file after line 34
Saving and Quitting	
:w	Save changes (write buffer)

Command	Meaning
:w **filename**	Write buffer to named file
:wq	Save changes and quit vi
ZZ	Save changes and quit vi
:q!	Quit without saving changes

6.17 SUMMARY

In this unit we have learn about the **vi editor,** is a powerful text editor available on almost all Unix systems. It is a very versatile text editor with many features that make it ideal for editing text files. The vi editor is a very user-friendly text editor with a wide variety of features that make it a great choice for editing text files. It is a very popular text editor used by many people, including programmers, web developers, and system administrators. We try to discuss on Introduction to vi editor,modes, status line commands, Opening & modifying a file, Saving a file and exiting vim, Search and Replace, undoing changes, yanking, Accessing multiple files, Window Commands, Interacting with system, Macros, vim configuration basics, syntax of ex commands, Addresses, Address symbols, options

6.18 CHECK YOUR PROGRESS

- **What do you mean by vi mode?**

  ……
  ……
  ……
  ……
  ……
  ……

- **What is ex command?**

  ……
  ……
  ……
  ……
  ……
  ……

- **How you can handle the multiple file in vi editor**

  ……
  ……
  ……
  ……
  …….

- **Discuss about file operation.**

  ……
  ……
  ……
  ……
  …….

6.19 FURTHER READING

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education
- https://www.redwood.com/article/unix-job-scheduling/
- https://www.geeksforgeeks.org/

UNIT-7 UNDERSTANDING PROCESSES IN UNIX

7.1 INTRODUCTION

A process is the active execution of a program and encompasses various components such as data from files, user inputs, and program instructions. In Unix/Linux systems, a new process is created every time an application is started, a program is run, or a command is executed. While each program or command generates a single process, an application may spawn multiple processes to handle different tasks.

7.1a. Objectives:

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of Processes in UNIX operating system
- process fundamentals,
- Types of process
- How to Initializing a process
- Process creation
- Concept of killing processes,
- managing multiple processes,

- changing process priority,
- scheduling of processes: at, batch, crontab command

7.2 WHAT IS A PROCESS?

A Process is a program under execution in a UNIX .

As a multitasking operating system, Unix can run multiple programs or applications simultaneously. Each running program is referred to as a "process" in Unix. Each process is assigned a unique process ID (PID), which is a numeric identifier. The first process to start when the system boots is assigned PID 1, with subsequent processes receiving incrementally higher numbers.

Each process is typically associated with a user—the one who initiated it. Additionally, processes may have a "controlling terminal," which is the virtual terminal from which the process was started and to which its input and output are connected.

7.3 TYPES OF PROCESSES

There are various types of processes:

1. **Parent and Child Process:**

   A **child process** is a Linux process that is created by another process known as the **parent process**. A child process can be created using one of two methods. The first method is the **fork system call**, which is often used in Unix-like operating systems, whereas the second method is the spawn method, which is preferred in the modern NT kernel of Microsoft Windows. A process is considered a parent process if it creates one or more child (sub process) processes.

2. **Zombie and Orphan process:**

   A **zombie process** is a Linux process that has finished its execution but still has an entry in the process table. Zombie processes usually get created when a child process completes its execution but the parent process has yet to read its exit status.

   An **orphan process** is a process that continues to run even after the parent process has been completed or terminated. An orphan process can be intentionally or unintentionally created.

3. **Daemon process:**

   Daemon processes are system-related background processes. These processes often run the permissions of root and service requests from other processes. Daemon processes often run in the background, waiting for specified events or tasks to occur. Daemon processes, unlike regular processes, do not require a controlling terminal to operate.

7.4 INITIALIZING A PROCESS

A process can be run in two ways:

1. **Foreground Process** : Every process when started runs in foreground by default, receives input from the keyboard and sends output to the screen. When issuing pwd command

   $ ls pwd

   Output:

   $ /home/uprtou/root

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

2. **Backround Process :** It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in background since they do not have to wait for the previous process to be completed. Adding & along with the command starts it as a background process

3. $ pwd &

   Since pwd does not wants any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter, : Output:

   [1] + Done pwd

   $

   That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully.

7.5 TRACKING ONGOING PROCESSES

ps can be used to see/list all the running processes.

$ ps

PID	TTY	TIME	CMD
19	pts/1	00:00:00	sh
24	pts/1	00:00:00	ps

For more information -f (full) can be used along with ps

$ ps –f

UID	PID	PPID	C	STIME	TTY	TIME	CMD
52471	19	1	0	07:20	pts/1	00:00:00f	sh
52471	25	19	0	08:04	pts/1	00:00:00	ps -f

For a single process information, ps along with process id is used

$ ps 19

PID	TTY	TIME	CMD
19	pts/1	00:00:00	sh

For a running program (named process) Pidof finds the process id's (pids)

Fields described by ps are described as:

UID: User ID that this process belongs to (the person running it)

PID: Process ID

PPID: Parent process ID (the ID of the process that started it)

C: CPU utilization of process

STIME: Process start time

TTY: Terminal type associated with the process

TIME: CPU time taken by the process

CMD: The command that started this process

There are other options which can be used along with ps command:

-a: Shows information about all users

-x: Shows information about processes without terminals

-u: Shows additional information like -f option

-e: Displays extended information

7.6 STOPPING A PROCESS

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it's pid is known.

$ ps –f

UID	PID	PPID	C	STIME	TTY	TIME	CMD
52471	19	1	0	07:20	pts/1	00:00:00	sh
52471	25	19	0	08:04	pts/1	00:00:00	ps –f

$ kill 19 Terminated

If a process ignores a regular kill command, you can use kill -9 followed by the process ID .

$ kill -9 19

Terminated

7.7 LISTING PROCESSES - THE PS COMMAND

You can use the ps command to list processes. Here is a sample process listing:

yt:~> ps

PID	TTY	STAT	TIME	COMMAND
12539	p3	S	0:00	-csh
12835	p0	S	0:00	-csh
13072	p4	S	0:00	–csh
13141	p5	SW	0:00	(tcsh)
13198	p3	S	0:01	vim processes.sgml
13203	p6	S	0:00	-csh
13204	p6	R	0:00	ps

The ps without any switches or other arguments shows processes which are being run by you, which have a controlling terminal. (We'll find out about processes with no controlling terminal later.) The PID listed is the "process ID" -- a numeric identifier which uniquely identifies each process running on the system.

Command line switches to ps

The command line switches to ps vary significantly between different flavours of Unix. However, since most systems support BSD style switches, those are the ones which will be described here.

Switch	Meaning
l	long format
a	show all processes (not just your own)
u	user format; shows username and start time
x	show processes with no controlling terminal
w	wide listing (ww makes it even wider)

Table : BSD-style command line switches for ps

The following process listing format is very common, but tends to produce a lot of output. Pipe its output through less or some other filter to ensure that you can see important information.

```
% ps auxw
USER       PID %CPU %MEM  SIZE   RSS TTY STAT START   TIME COMMAND
daemon     150  0.0  0.0  1104     0  ?  SW  Oct 25   0:00 (portmap)
mail     14902  0.0  0.0  1712     0  ?  SW  Oct 26   0:01 (sendmail)
root         1  0.0  0.1  1016    72  ?  S   Oct 25   0:03 init [2]
root         2  0.0  0.0     0     0  ?  SW  Oct 25   0:13 (kflushd)
root         3  0.0  0.0     0     0  ?  SW  Oct 25   0:11 (kupdate)
root         4  0.0  0.0     0     0  ?  SW  Oct 25   0:00 (kpiod)
root         5  0.0  0.0     0     0  ?  SW  Oct 25   0:31 (kswapd)
root       120  0.0  0.5  1348   212  ?  S   Oct 25   0:05 /sbin/syslogd
root       122  0.0  0.0  1288     0  ?  SW  Oct 25   0:00 (klogd)
root       142  0.0  0.1   996    48  ?  S   Oct 25   0:00 /sbin/kerneld
root       154  0.0  0.0  1304     0  ?  SW  Oct 25   0:00 (inetd)
root       182  0.0  0.1  1020    56  ?  S   Oct 25   0:02 /usr/sbin/gpm -
m /dev/psaux -t ps2 -l "a-zA-Z0-9_.:~/\300-\326\330-\366\370-\377"
root       245  0.0  0.0  1904     0  ?  SW  Oct 25   0:00 (safe_mysqld)
root       252  0.0  0.1  4040    44  ?  S N Oct 25   0:00 (mysqld)
root       261  0.0  0.1  2180    52  ?  S   Oct 25   0:01 /usr/sbin/apache
root       272  0.0  0.0  1136     0   2 SW  Oct 25   0:00 (getty)
```

The most popular Unix variant which uses completely different switches for ps is Solaris. To find out about the command line switches on that (or any other) flavour of Unix, type man ps. Note that a BSD-style ps is available on Solaris systems in /usr/ucb/ps

Process Creation:

In Unix and POSIX systems, creating a new process typically involves calling fork() followed by exec(). When you use fork(), it duplicates the current process, including its data, code, environment variables, and open files. This results in a child process that is a near-identical copy of the parent process, with a few exceptions.

The fork() system call returns a process ID (PID) to the parent process, which is the PID of the newly created child process, while the child process receives a return value of zero. Both processes continue executing from the point where fork() was called.

If the intention is to run a different program, the child process can then call exec() to replace its current code and data with those of a new program. The new child process will retain the open files inherited from the parent, with the exception of those files whose close-on-exec flag was set using fcntl.

In pseudo-code, this is:

```
result := fork()
 if result < 0
 #fork() failed
 elseif result > 0
 #this is the Parent
 elseif
 #result=0, this is the Child
 #call exec() to "replace myself" with another prog
 exec()
 #control will Never return here from exec()
 #instead if goes to the new program code
 endif
```

Killing processes

You can kill any process which you own (but not those owned by other users) by using the kill command:

```
% ps
PID   TTY   STAT   TIME   COMMAND
12539 p3  SW     0:00    (tcsh)
13141 p5   SW    0:00    (tcsh)
13198 p3   S      0:07    vim processes.sgml
13279 p6   R      0:00   ps
% kill     13198
```

% ps

PID TTY STAT TIME COMMAND

12539 p3 SW 0:00 (tcsh)

13141 p5 SW 0:00 (tcsh)

13279 p6 R 0:00 ps

You can send other signals to processes using the kill command's optional switches. You can give an signal name as an optional switch, or a signal number. Some of the common signal names and numbers are listed in the manual page for kill, or you can get a list of available signal names with kill -l

% kill -l

HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH POLL PWR UNUSED

The most commonly used signal is KILL, which forces a process to be killed even if a normal kill command cannot kill it. This is often useful if a program has "hung" and will not respond to anything. The signal number for KILL is 9, therefore the following two commands are equivalent:

% kill -KILL 13198

% kill -9 13198

7.10 OTHER PROCESS COMMANDS

Linux has many commands for handling and interacting with processes. Here are some popular Linux process commands:

1. **top:**
   This command is used to show all the ongoing Linux processes within the working environment of Linux.

   Syntax:

   top

2. **bg:**
   This is a job control command that resumes a stopped job while keeping it running in the background.

   Syntax:

   bg [job]

   For example

   bg %19

3. **fg:**
   This is a job control command that resumes a stopped job while keeping it running in the foreground.

   Syntax:

   fg [%job_id]

   For example

fg 19

4. **nice:**
   The nice command is used to start a new process while specifying a priority (nice) value. The nice value goes from -20 to 19, with -20 being the highest priority.

   Syntax:

   nice [-nice value]

   renice : To change the priority of an already running process renice is used.

   Syntax:

   renice [-nice value] [process id]

5. **df:**
   This command is used to display the free disk space(Hard Disk) on all the file systems.

   Syntax:

   df

6. **free:**
   This command is used to display the total amount of free and used memory (RAM) on the Linux system. **Syntax:**

   free

7.11 RUNNING A PROCESS IN THE BACKGROUND

On Unix, it is possible to run processes in the background. This means that the process becomes detached from its controlling terminal and runs more-or-less invisibly to the user.

To start a process running in the background, simply put an ampersand (&) after the command on the command line.

% cp -r * /tmp &

[1] 13319

Listing running jobs

To get a listing of running jobs, use the **jobs** command.

% jobs

[2] + Suspended (tty output) top

[3] - Running tar -c /www > www.tar

Note: a job is not the same as a process. A job may have many different processes associated with it. For instance, if you create a pipeline on the command line (that is, connect multiple programs together using pipes) there will be several processes running, but they constitute only one job.

7.12 BRINGING A BACKGROUNDED PROCESS TO THE FOREGROUND

As you can see, when you run a program in the background it will tell you the process ID it has been given, and also a job number.

Note: The Bourne Shell (sh) does not have job control; this section assumes you are using a more featureful shell such as bash, tcsh, csh or ksh.

This job number allows you to bring a job to the foreground using the fg command. The following command would bring job number 1 to the foreground:

% fg %1

Back grounding an already-running job

Lastly, if you wish to send an already-running program to the background, you can do this by first pressing CTRL-Z to suspend it, then typing the **bg** command:

% yt:~/work/training/unixtools> tar -c /www > www.tar

(CTRL-Z is pressed at this point)

Suspended

% bg

[3] tar -c /www > www.tar &

7.13 KILLING A RUNNING JOB

You can also kill a running job with the kill by giving it the job number as an argument:

% kill %3

[3] - Terminated tar -c /www > www.tar

7.14 SCHEDULING OF PROCESS

There are various ways for scheduling of processes

1. **at Command**

   The at command is a Unix/Linux system utility that allows users to schedule jobs or commands to run at a scheduled time in the future and provides a simple command-line interface for job scheduling and can be used for one-time or recurring jobs.

   For example, to schedule a job to be executed at 3:00 PM, you can use the following command:

   $ at 3pm

   This will open the at command prompt. You can then enter the command you want to execute at 3:00 PM. For example, to create a file named testfile.txt, you can enter:

   $ echo "Hello, World!" > testfile.txt

   After you have entered the command, press Ctrl + D to exit the prompt. The job will be scheduled and executed at the specified time.

   You can also specify the date and time using the following format:

   $ at <time> <date>

   For example, to schedule a job to be executed on January 1, 2024 at 9:00 AM, you can use the following command:

   $ at 9am 01/01/2024

   The at command is useful for scheduling Unix jobs, especially for one-time or ad hoc tasks. However, there may be better solutions for recurring or complex tasks requiring more flexibility and scheduling control.

2. The batch Command

The batch command operates similarly to the at command, but with three significant differences:

1. You can only use the batch command interactively.
2. Rather than scheduling jobs to execute at a specific time, you add them to the queue, and the batch command executes them when the system's average load is lower than 1.5.
3. Due to the above, you never specify a date and time with the batch command.

   When you use the batch command, you call it by name with nocommand line parameters like so:

   batch

```
dave@howtogeek:~$ batch
warning: commands will be executed using /bin/sh
at> ./cloud-backup.sh
at> <EOT>
job 16 at Wed Dec 18 09:08:00 2019
dave@howtogeek:~$
```

3. Crontab command

The cron utility is a time-based job scheduler in Unix-like operating systems. It runs in the background and regularly checks the system's crontab files for scheduled jobs. These jobs can be added using the crontab command, which allows users to specify the schedule and command to run in a specific format.

To add a new job to your crontab, open a terminal or shell prompt and type crontab -e to open the crontab editor.

From there, you can specify the schedule and command to run using five asterisks to represent the minute, hour, day of the month, month, and day of the week when the command should run.

The **cron** daemon runs in the background and regularly checks the system's crontab files to determine which jobs to run and when to run them.Here's an example of how to run a unix job using cron utility:

1. Open a terminal or shell prompt.
2. Type crontab -e to open the crontab editor.
3. Add a new job by specifying the schedule and command to run in the following format: * * * * * command
4. The five asterisks represent the minute, hour, day of the month, month, and day of the week when the command should run. For example, 0 9 * * 1-5 means the command should run at 9:00 AM on weekdays.
5. Save and exit the editor. The new job will be added to the crontab.

You can also view the current list of jobs in your crontab by typing crontab -l. To remove a job, use **crontab -r.**

The crontab files are located in the **cron.d** and **cron.daily** directories, and they can be edited using a variety of Linux/unix commands. Log files can be used to track the execution of cron jobs and any errors that occur during the process

7.15 SUMMARY

In this unit, we have learn about the basic concept of Processes in Unix, process fundamentals, connecting processes with pipes, Redirecting input output, manual help, Background processing, killing processes, managing multiple processes, changing process priority, scheduling of processes: at, batch, crontab command

7.16 CHECK YOUR PROGRESS

- **What do you mean by processes?**

  ……
  ……
  ……
  ……
  ……

- **How you Kill off your backgrounded vi session using kill?**

  ……
  ……
  ……
  ……
  ……

- **Discuss the major process related commands**

  ……
  ……
  ……
  ……
  ……
  ……

- **Discuss about process creation.**

  ……
  ……
  ……
  ……
  ……

7.17 FURTHER READING

- Unix and shell programming by B.M. Harwani, OXFORD university press
- UNIX : Concepts and Applications by Sumitabha Das TMH Education
- https://www.redwood.com/article/unix-job-scheduling/
- https://www.geeksforgeeks.org/

BLOCK-3

INTRODUCTION

This block is a course on "**Shell programming** in Unix ". This block course contains three units. Unit 1 is an introductory unit on **Shell programming**. With this unit, the learners will be acquainted with some important concepts **of Shell programming** like Basic of shell programming, various types of shells, shell programming in bash, Shell variables, shell keywords, assigning values to variables, unchanging variables, positional parameters, passing command line arguments, arithmetic in shell scripts, read, echo command etc,. Unit 2 discusses the Working with **Decision Control Structures**. This includes the Decision statements, if, else, elif, test command, nested if-else, forms of if, use of logical operators, case control structure . Unit 3. is the last unit and it discusses about **Loop control structures** of Unix.and cover the topic lke looping statements: while, until, for, reading from a file, using with command line arguments, nested loops, break statement, continue statement

The learners will learn how to work with Unix **Shell programming** . The units of this block also discusses abut **Shell programming** , Working with **Shell programming** Commands of Unix While going through a unit, you will notice some boxes along-side, which have been included to help you know some of the difficult, unseen terms. Some "ACTIVITY' (s) have been included to help you apply your own thoughts. Again, we have included some relevant concepts in "LET US KNOW" along with the text. And, at the end of each section, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be better if you solve the given problems in these boxes immediately, after you finish reading the section in which these questions occur and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

UNIT-8 BASICS OF SHELL PROGRAMMING

Structure

8.1 INTRODUCTION

Every Unix system includes at least one shell, which is a program that interfaces between the user and the kernel, and thus the hardware. The shell acts as a command interpreter, allowing users to enter Unix commands. Upon successful boot-up, the shell displays a command line prompt, often denoted by symbols like $ or %, where users can input commands.

A shell script is a text file that contains a sequence of Unix commands. It's called a script because it allows multiple commands to be executed in sequence, rather than typing each command individually. Each shell has its own scripting language with unique syntax, variables, and control flow features. For instance, bash scripting is a widely used form of shell scripting. Shell scripting is particularly suited for tasks related to the shell environment, such as creating command pipelines, redirecting outputs to files, and reading from standard input. This specialization makes it more straightforward for these tasks compared to other scripting languages. For example, running the cd command in Python would require importing the os module and using its chdir function.

8.1 OBJECTIVES

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of Shell programming
- various types of shells,
- shell programming in bash,
- Shell variables,
- Shell keywords,
- Assigning values to variables, unchanging variables,
- Positional parameters,
- Passing command line arguments,
- arithmetic in shell scripts and read, echo command

8.2 TYPES OF SHELLS

There are several types of shells available in Unix and Unix-like operating systems. Some of the most common ones include:

1. **Bash (Bourne Again Shell)**: This is one of the most popular shells, widely used for its powerful features and compatibility with the Bourne shell (sh). It includes advanced scripting capabilities, command-line editing, and a large array of built-in functions.
2. **sh (Bourne Shell)**: The original Unix shell, created by Stephen Bourne. It's known for its simplicity and scripting capabilities. Many modern shells are compatible with sh, and it is often used for writing shell scripts.
3. **csh (C Shell)**: Created by Bill Joy, this shell offers a syntax similar to the C programming language, which can make it easier for programmers familiar with C. It includes features like job control and command history.
4. **tcsh (TENEX C Shell)**: An enhanced version of csh, tcsh adds additional features such as command-line editing and filename completion, making it more user-friendly.
5. **zsh (Z Shell)**: Known for its rich feature set and configurability, zsh includes features like spell checking, improved tab completion, and powerful customization options. It is highly regarded for interactive use and scripting.
6. **ksh (Korn Shell)**: Developed by David Korn, this shell combines features from both the Bourne shell and the C shell. It offers features like associative arrays and floating-point arithmetic, making it suitable for complex scripting.
7. **fish (Friendly Interactive SHell)**: Designed to be user-friendly and easy to use, fish emphasizes interactive features, such as syntax highlighting and autosuggestions. It aims to improve the user experience with intuitive defaults and a focus on usability.

Each of these shells has its own strengths and is suited to different types of tasks, so users can choose the one that best fits their needs and preferences.

8.3 SHELL PROGRAMMING IN BASH

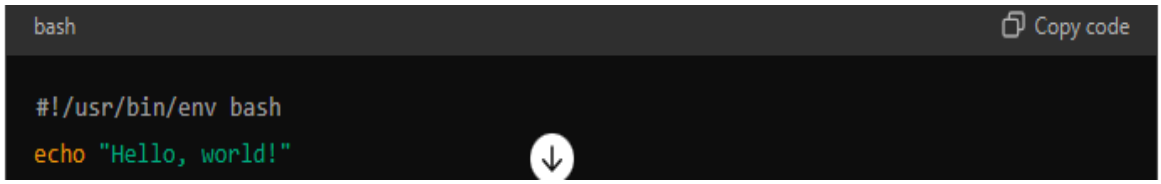
Bash scripting involves writing scripts specifically for the Bash shell (Bourne Again SHell). To determine which shell you are currently using, you can execute the command ps -p $$. On Unix/Linux systems, Bash is often the default shell. However, if you are using macOS or Windows, you might need to switch to Bash manually.

On macOS, you can start a Bash shell by running exec bash. On Windows, you can initiate a Bash shell by running bash.

First Script

Now that you have access to a Bash shell, let's write your first Bash script, named hello.sh. Shell scripts typically use the .sh extension to denote that they are scripts. To execute the script, you can use the command sh hello.sh.

Here's a simple hello.sh script:



Explanation:

1. #!/usr/bin/env bash: This is the **shebang** line, which tells the system to use the Bash shell to interpret the script.
2. echo "Hello, world!": This command outputs the text "Hello, world!" to the terminal.

To run this script:

1. Save the above content to a file named hello.sh.
2. Make the script executable with the command: chmod +x hello.sh.
3. Execute the script by typing: ./hello.sh.

Get smarter responses, upload files and images, and more.

Shebangs:

The first line of a script, #!/usr/bin/env bash, is known as the shebang (or hashbang). This line starts with a # (pound symbol) followed by an ! (exclamation mark). The shebang specifies which interpreter should be used to execute the script. In this case, it indicates that the script should be run with the Bash interpreter.

Key Points About Shebang:

- **Purpose**: The shebang line directs the system to use the specified interpreter for the script. For example, #!/usr/bin/env bash tells the system to use Bash, which is found using the env command.
- **Portability**: Using #!/usr/bin/env bash is recommended for better portability. This approach allows the script to work on systems where Bash might not be located in the default /bin directory. Instead, env locates Bash wherever it is installed on the system.

- **Alternative**: An alternative shebang line is #!/bin/bash. This directly specifies the location of the Bash interpreter but may not be as portable if Bash is installed in a different location.

If you wanted to use a different shell, such as zsh, you would change the shebang line accordingly, for example, #!/usr/bin/env zsh.

By using #!/usr/bin/env bash, you ensure that your script will run with the Bash interpreter found in the user's PATH, making it more adaptable to various environments.

8.4 HOW TO WRITE SHELL SCRIPT

Following steps are required to write shell script:

(1) Use any editor like vi or mcedit to write shell script.

(2) After writing shell script set execute permission for your script as follows syntax:

chmod permission your-script-name

Examples:

```
$ chmod +x your-script-name
$ chmod 755 your-script-name
```

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

(3) Execute your script as syntax:

bash your-script-name sh your-script-name

./your-script-name

Examples:

```
$ bash bar
$ sh bar
$ ./bar
```

NOTE In the last syntax ./ means current directory, But only . (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for . (dot) command is as follows

Syntax:

. command-name

Example:

```
$ . foo
```

Now you are ready to write first shell script that will print "Knowledge is Power" on screen. See the common vi command list , if you are new to vi.

```
$ vi first
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

This will not run script since we have not set execute permission for our script first; to do this type command

```
$ chmod 755 first
$ ./first
```

First screen will be clear, then Knowledge is Power is printed on screen.

Script Command(s)	Meaning
$ vi first	Start vi editor
# # My first shell script # 	# followed by any text is considered as comment. Comment gives more information about script, logical explanation about shell script. Syntax: # comment-text
clear	clear the screen
echo "Knowledge is Power"	To print message or value of variables on screen, we use echo command, general form of echo command is as follows syntax: echo "Message"

How Shell Locates the file (My own bin directory to execute script)

For shell script file try to give file extension such as .sh, which can be easily identified by you as shell script.

Exercise:

```
$ vi ginfo
#
#
# Script to print user information who currently login , current date
& time
#
clear
echo "Hello $USER"
echo "Today is \c ";date
echo "Number of user login : \c" ; who | wc -l
echo "Calendar"
cal
exit 0
```

8.5 Running a Script

You can run a shell script in a couple of different ways:

1. **Using a Shell Interpreter**: You can execute the script by specifying the shell interpreter directly. For example:
    - sh hello.sh — Runs the script using the sh shell.
    - bash hello.sh — Runs the script using the bash shell.
    - zsh hello.sh — Runs the script using the zsh shell.
2. **Making the Script Executable**: Alternatively, you can make the script itself executable and then run it directly. Here's how you do it:
    - **Make the Script Executable**: Use the chmod (change mode) command to add execute permissions to the script:

This command changes the file's permissions to make it executable.

- **Run the Script**: After setting the executable permissions, you can run the script directly by prefacing it with ./:

```bash
./hello.sh
```

This approach is similar to running other executable programs.

By using these methods, you can execute your shell scripts efficiently, either by directly invoking the interpreter or by making the script an executable file.

8.6 Variables and Strings

Variables

Now that we have a basic script, let's dive into the mechanics of bash scripting. When creating a bash script, you assign variables using the syntax x=foo. To access the value of this variable, you use $x. It's important to avoid adding extra spaces around the = sign when assigning a variable. For instance, writing x = foo would cause the shell to treat it as running a command named x with = and foo as arguments. In general, the shell treats spaces as delimiters between arguments.

Strings

We can also define strings in a bash script. If we want to define a string literal, we will use single quotation marks: '$x' . If we want to define a string that allows substitution, we will use double quotes: "$x" . The double quotes will allow for substitution of variables:

```
1 x= foo
2 echo '$x '
3 # prints $x
4 echo "$x"
5 # prints foo
```

8.7 Some Important Keywords use in shell scripts:

In shell scripting, certain keywords and reserved words are used to control the flow of execution, define variables, and handle various tasks. Here's a summary of common keywords used in shell scripts:

Control Flow Keywords

1. **if / then / else / elif / fi**: Used for conditional execution.

Example:

```
if [ condition ]; then
  # commands
```

```
elif [ another_condition ]; then
  # commands
else
  # commands
fi
```

2. **case / esac**: Used for multi-way branching based on pattern matching.
    - Example:

```
case "$variable" in
  pattern1) # commands ;;
  pattern2) # commands ;;
  *) # default commands ;;
esac
```

3. **for / in / do / done**: Used for looping through a list of items.

    Example:

```
for item in list; do
  # commands
done
```

4. **while / do / done**: Used for looping while a condition is true.
    - Example:

```
while [ condition ]; do
  # commands
done
```

5. **until / do / done**: Used for looping until a condition becomes true.
    - Example:

```
until [ condition ]; do
  # commands
done
```

Variable and Parameter Keywords

6. **export**: Used to set environment variables.
    - Example:

```
export VAR_NAME=value
```

7. **unset**: Used to remove variables or functions.
    - Example:

```
unset VAR_NAME
```

Function Keywords

8. **function**: Used to define a function (though not always necessary, as functions can be defined without this keyword).

   Example:

   ```
   function my_function() {
     # commands
   }
   ```

9. **return**: Used to return a value from a function.

   Example:

   ```
   function my_function() {
     return 1
   }
   ```

Special Keywords

10. **break**: Exits from the nearest enclosing loop.

    Example:

    ```
    while true; do
      if [ condition ]; then
        break
      fi
    done
    ```

11. **continue**: Skips the remaining commands in the current iteration of the loop and proceeds to the next iteration.
    - Example:

      ```
      for i in 1 2 3; do
        if [ $i -eq 2 ]; then
          continue
        fi
        echo $i
      done
      ```

12. **exit**: Exits from the script or function with a specific exit status.
    - Example:

      ```
      if [ condition ]; then
        exit 1
      fi
      ```

13. **set**: Used to set shell options and positional parameters.

- Example:

  set -x # Enable debugging

14. **shift**: Shifts positional parameters to the left.
    - Example:

      shift 2 # Shift the positional parameters by 2

Miscellaneous Keywords

15. **source**: Executes commands from a file in the current shell.
    - Example:

      source script.sh
16. **readonly**: Makes variables read-only, preventing modification.
    - Example:

      readonly VAR_NAME=value

These keywords help structure and control the behaviour of shell scripts, enabling complex logic and operations to be performed efficiently

8.8 Assigning values to variables

Variable Names:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_). By convention, Unix shell variables will have their names in UPPERCASE. It is case sensitive. you cannot use special characters.

Types of variables

There are Two types of variables in unix :

- ✓ User defined variables
- ✓ system/environmental varisbles.

The following examples are valid variable names –

_ALI

TOKEN_A

VAR_1

VAR_2

Following are the examples of invalid variable names –

2_VAR

-VARIABLE

VAR1-VAR2

VAR_A!

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

User defined variables:

A variable defined by user known as user defined variables.

Defining Variables :

Variables are defined as follows −

variable_name=variable_value/expression/command

For example −:

$myvar = 1234 #No space on either side of equal sign

By default any variable created in shell are of string type. so, the values are stored in ASCII rather than binary. Variables created in shell are automatically removed as soon as shell is expired. when shell reads the command line, it interprets any word preceded by $ as a variable and replace the word by the value of the variable.

To display content , a user can use echo command:

$echo $myvar <enter>

ans: 1234

$

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

VAR1="seems shah"

VAR2=100

Accessing Values :

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

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

#!/bin/sh

NAME=" seems shah "

echo $NAME

The above script will produce the following value −

seems shah

Read-only Variables :

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

For example, the following script generates an error while trying to change the value of NAME

#!/bin/sh

NAME=" seems shah "

readonly NAME

NAME="riya"

The above script will generate the following result −

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

8.9 Shell's interpretation at prompt:

when user enters a command on command line shell perform some internal steps before the command directs its execution to a utility or s program. Following are the steps:

1. **PARSING**: in this stage, the shell divides the command line into words, if it is not quoted or escaped. The shell uses system variable IFS (internal field separator) to delimiter the value of word delimiter, which is space and tab.

   When shell encounters 2 or more spaces or tabs in the command line then they are replaced with a single space.

   for example, a=10

   $echo value of variable a=$a <enter> //multiple spaces

   ↓ Parsing

   $ echo value of variable a=$a //spaces are replaced.

2. **Variable Evaluation**: in this stage, the shell is looking for variable names. for example, a=10

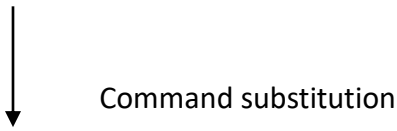
   $echo value of variable a=$a <enter> //$a is consider as variable

   ↓ Variable evaluation

   $ echo value of variable a=10 //value is assigned.

3. **Command Substitution**: Any Unix command enclosed within back quotes is executed by the shell and its output is substituted as argument for another command into a command line.

E.g.: echo “Today’s date: 'date'" // date command is enclosed within back quote"



echo "Today's date : MON Feb 24 14:03:03 IST 2024" // The output of date command is become an argument for echo command.

4. **Redirection:** in this stage, shell scans for the characters >,< and >> and open a file associated with this characters in a particular mode.

e.g.: wc <f1 here, shell opens a file instead of command and display number of lines, words and character in file f1.

5. **Wild-card interpretation**: A number of characters are interpreted by the Unix shell before any other action takes place. These characters are known as wildcard characters.

Usually these characters are used in place of filenames or directory names.

* An asterisk matches any number of characters in a filename, including none.

? The question mark matches any single character.

[ ] Brackets enclose a set of characters, any one of which may match a single character at that position.

- A hyphen used within [ ] denotes a range of characters.

~ A tilde at the beginning of a word expands to the name of your home directory. If you append another user's login name to the character, it refers to that user's home directory.

Here are some examples:

1. cat c* displays any file whose name begins with c including the file c, if it exists.
2. ls *.c lists all files that have a .c extension.
3. cp ../rmt?. copies every file in the parent directory that is four characters long and begins with rmt to the working directory. (The names will remain the same.)
4. ls rmt[34567] lists every file that begins with rmt and has a 3, 4, 5, 6, or 7 at the end.
5. ls rmt[3-7] does exactly the same thing as the previous example.
6. ls ~ lists your home directory.
7. ls ~hessen lists the home directory of the guy1 with the user id hessen.

6. **PATH evaluation**: finally, it examine the system variable PATH to search a command into the sequence of directories stored in it. If it is found then load an executable of command into memory and then execute it otherwise it sows error message on standard output.

8.10 Assigning values to variables:

In Bash, to assign a value to a variable, you use the '=' operator with the syntax VAR=Value. It's important to note that there should be no spaces between the variable name, the '=' operator, and the value being assigned. Here's how you set a variable correctly in Bash:

VAR=Value

For example:

name="John"

count=10

filename="example.txt"

In these examples:

- name, count, and filename are variable names.
- "John", 10, and "example.txt" are the values assigned to these variables.

It's crucial to avoid placing spaces around the '=' sign when assigning variables in Bash. This strict adherence ensures that Bash interprets the statement correctly as an assignment operation rather than treating the spaces as part of the variable name or value. Following this syntax helps prevent syntax errors and ensures consistent behavior in your Bash scripts

The advantages of using variables in Bash are numerous. They allow you to store and manipulate data, making your scripts more flexible and efficient. However, there are potential pitfalls to be aware of. One common mistake is adding spaces around the '=' operator when assigning values to variables. In Bash, this will result in an error.

Here's what happens when you add spaces around the '=' operator:

```
myVar = 'Bash Beginner'


# Output:
# myVar: command not found
```

As you can see, Bash interprets myVar as a command, which is not what we intended. So, remember, no spaces around the '=' operator when setting variables in Bash!

8.11 Unchanging variables

In some applications, there is a requirement for variables to maintain a constant or fixed value. For example, if we want the variable a to always remain set at 20 and not be altered, we can achieve this by declaring:

a=20

In Bash scripting, this assignment statement assigns the value 20 to the variable a. Once assigned, a will retain the value 20 throughout the script execution unless explicitly modified elsewhere in the script. This approach ensures that a behaves as a constant within the script, maintaining consistency and predictability in its usage.

Example #1

```
$a = 20
```

```
$readonly a
```

To create read-only variables in Bash, preventing their values from being changed once set, you can use the readonly command. Here's how you can do it:

```
readonly variable_name
```

For example, if you want to make the variable a read-only after setting it to 20:

```
a=20
readonly a
```

After this declaration, any attempt to change the value of a will result in an error:

```
a=30  # This will produce an error: "readonly variable"
```

To unset or clear a variable from the shell environment, you can use the unset command:

```
unset variable_name
```

For instance, to unset the variable a:

```
unset a
```

After unsetting, the variable a will no longer exist in the current shell session. These commands (readonly and unset) provide flexibility in managing variables and maintaining script integrity in Bash.

Example #2

```
$a = 20
```

```
$echo a
```

```
20
```

```
$unset a
```

```
$echo a  // a have no value now so it prints nothing
```

2. Echo keyword

To print either the value of any variable or words under double quotation.

Example #1

```
x=20
```

```
echo $x
```

Output:

Example #2

```
echo "Hello World!"
```

Output: Hello world!

8.12 Positional parameters:

In computing, a variable is a named entity that stores a value, such as a number, text, filename, or other data type. In Bash scripts, variables are denoted by a preceding dollar sign ($). For example, consider our variable named $variable.

On the other hand, a positional parameter in Bash refers to an argument provided when executing a script. These parameters are stored in special variables managed by the shell. Unlike variables defined within a script, positional parameters are passed from outside the script, offering greater flexibility as they can be altered without script modification.

In a script execution context, $0 represents the command itself, while subsequent arguments are accessed using $1, $2, and so forth. Conventionally, when discussing positional parameters, $0 is ignored, and counting starts with $1.

Positional parameters in shell scripts are separated by spaces. Each segment following a space is treated as an individual parameter by the shell.

For instance, consider the command "mycommand one two three four". Here, "mycommand" is the command name, and it receives three parameters: "one", "two", and "three four".

During the execution of "mycommand", Bash assigns it the following positional parameters:

- $0: The name of the command itself, in this case, "mycommand".
- $1: The first parameter, "one".
- $2: The second parameter, "two".
- $3: The third parameter, "three four".

In this example, "three four" is treated as a single parameter because it is enclosed within quotes or is not separated by a space.

Variable name	Value
$0	mycommand
$1	one
$2	two
$3	three four
$#	3
$@	one two three four
$*	one two three four

In Bash scripting, several special variables are crucial for handling command line arguments:

- **$0**: Represents the command itself, typically the name used to invoke the script or program.
- **$1, $2, $3, ...**: These variables hold the values of the first, second, third, and subsequent parameters passed to the script or program. For instance, $1 holds the value of the first parameter, $2 holds the value of the second parameter, and so on.

- **${10}, ${123}, ...**: Parameters greater than 9 can be accessed using curly braces around the number. For example, ${10} refers to the tenth parameter, and ${123} refers to the 123rd parameter.
- **$#**: Contains the total number of positional parameters passed to the script or program, excluding $0.
- **$@**: Represents all the positional parameters passed to the script or program, excluding $0. When referenced as "$@", each parameter is treated as a separate entity.
- **$***: Similar to $@, but when enclosed in double quotes ("$*"), it expands to "$1c$2c$3...", where 'c' is the first character of the Internal Field Separator (IFS). By default, IFS includes spaces, tabs, and newlines, defining where word splitting occurs.

If IFS is set to "_", then "$@" expands to:

one two three four

While "$*" expands to:

one_two_three four

These distinctions are crucial for how Bash interprets and separates command line arguments and parameters when scripting or running programs in a Unix-like environment. Understanding these variables ensures accurate handling and manipulation of input within Bash scripts.

8.13 Passing command line arguments

A command line argument in Bash is a parameter supplied to a script or program at execution time. These arguments enable users to dynamically influence the behaviour of a script or control the output it produces.

To pass an argument to a Bash script, simply write the argument after the script's name when invoking it from the command line. For example:

```
./myscript.sh arg1 arg2 arg3
```

In this example:

- ./myscript.sh is the command to execute the script named myscript.sh.
- arg1, arg2, and arg3 are the command line arguments passed to the script.

Inside the Bash script, these arguments are accessed using special variables:

- $1 refers to the first argument (arg1 in this case).
- $2 refers to the second argument (arg2 in this case).
- $3 refers to the third argument (arg3 in this case), and so on.

These arguments allow scripts to be more versatile and adaptable to different inputs or scenarios without needing to modify the script itself.

Let’s see how this works using an example script:

```
#!/usr/bin/env bash
echo "The first fruit is: $1"
```

```
echo "The second fruit is: $2"
echo "The third fruit is: $3"
```

If we run our script and don't give an argument, we will see no output for our pre-defined variables:

```
./script.sh
The first fruit is:The second fruit is:The third fruit is:
```

Alternatively, if we provide three fruits, our script detects these and will return those values back to use via the pre-defined variables.

```
./fruit.sh apple pear orange
The first fruit is: appleThe second fruit is: pear The third fruit is: orange
```

Sometimes, we may want to access all of the arguments. We can do this using **$@**.

Let's update our example script to return all of the fruits provided as arguments to the script as well:

```
#!/usr/bin/env bash
echo "The first fruit is: $1"
echo "The second fruit is: $2"
echo "The third fruit is: $3"
echo "All fruits are: $@"
```

When we run our script, you can see that we now have an extra output which lists all of the fruits we gave to our script on the command line.

```
./fruit.sh apple pear orange
The first fruit is: appleThe second fruit is: pearThe third fruit is: orangeAll fruits are: apple pear orange
```

8.14 Arithmetic in Shell

In Bash, you can perform numeric operations on integer variables directly within scripts. Bash supports basic arithmetic operations such as addition, subtraction, multiplication, division, and modulus. Here's how you can carry out these operations:

Example:

```
# Define integer variables
a=10
b=5

# Addition
sum=$((a + b))
echo "Sum: $sum"

# Subtraction
```

```
difference=$((a - b))
echo "Difference: $difference"

# Multiplication
product=$((a * b))
echo "Product: $product"

# Division (integer division)
quotient=$((a / b))
echo "Quotient: $quotient"

# Modulus
remainder=$((a % b))
echo "Remainder: $remainder"
```

Explanation:

- **Integer variables**: a and b are defined as integers.
- **Arithmetic operations**: Inside ((...)), Bash evaluates the arithmetic expression and assigns the result to the respective variables (sum, difference, product, quotient, remainder).
- **Output**: Each result is echoed to the console.

Notes:

- Bash uses ((...)) for arithmetic evaluation.
- Arithmetic operations in Bash are performed on integers. Floating-point arithmetic is not directly supported without using external tools or commands.
- Variables used in arithmetic operations can be initialized directly or can hold values obtained from other commands or input.

These capabilities allow Bash scripts to handle basic numeric calculations effectively, supporting various tasks such as calculations, counters, and condition checks based on numerical values.

Operation	**Operator**
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	** (bash only)
Modulo	+ * / %

Arithmetic operations in bash can be done within the $((⋯)) or $[⋯] commands

- ✓ Add two numbers: $((1+2))
- ✓ Multiply two numbers: $[$a*$b]
- ✓ You can also use the let command: let c=$a-$b
- ✓ or use the expr command: c=‘expr $a - $b‘

In tcsh,

- ✓ Add two numbers: @ x = 1 + 2
- ✓ Divide two numbers: @ x = $a / $b
- ✓ You can also use the expr command: set c = ‘expr $a % $b‘

- **Note the use of space**

bash space required around operator in the expr command

tcsh space required between @ and variable, around = and numeric operators.

You can also use C-style increment operators

bash let c+=1 or let c- -

tcsh @ x - = 1 or @ x++

/=, *= and %= are also allowed.

bash

The above examples only work for integers.

- Using floating point in bash or tcsh scripts requires an external calculator like GNU bc.

✓ Add two numbers:

```
echo "3.8 + 4.2" | bc
```

✓ Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

✓ Call bc directly:

```
bc <<< "scale=5; 2/5"
```

✓ Use bc -l to see result in floating point at max scale:

```
bc -l <<< "2/5"
```

✓ You can also use awk for floating point arithmetic

Syntax:

expr op1 math-operator op2

Examples:

```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 \* 3
$ echo `expr 6 + 3`
```

8.15 The read Statement

In Bash scripting, you can use the read command to obtain input from the user via the keyboard and store it in a variable. Here's how you can do it:

```
#!/bin/bash

# Prompt the user to enter their name
echo "Please enter your name:"
# Use the read command to capture input from the user and store it in the variable 'name'
read name

# Display a greeting using the input stored in the 'name' variable
echo "Hello, $name! Welcome to Bash scripting."
```

Explanation:

1. **echo "Please enter your name:"**: This line prints a message prompting the user to enter their name.
2. **read name**: The read command reads a line from the standard input (usually the keyboard) and assigns it to the variable name.
3. **echo "Hello, $name! Welcome to Bash scripting."**: This line uses the value stored in the name variable to display a personalized greeting.

Usage:

- Save the above script to a file (e.g., input_example.sh).
- Make the script executable with chmod +x input_example.sh.
- Run the script with ./input_example.sh.
- Enter your name when prompted and press Enter.

The read command is versatile and can be used to capture various types of input (strings or numbers) from the user interactively within a Bash script. Adjustments can be made to handle different types of input or to provide different prompts based on the script's requirements.

Syntax:

read variable1, variable2,...variableN

8.16 echo Command

Use echo command to display text or value of variable.

echo [options] [string, variables...]

Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)

\b backspace

\c suppress trailing new line

\n new line

\r carriage return

\t horizontal tab

\\ backslash

For e.g. $ echo -e "An apple a day keeps away \a\t\tdoctor\n"

8.17 SUMMARY

In this unit, we have learn about the basic concept of shell programming, various types of shells, shell programming in bash, Shell variables, shell keywords, assigning values to variables, unchanging variables, positional parameters, passing command line arguments, arithmetic in shell scripts, read, echo,

8.18 Check your progress:

- **What do you mean by shell script?**

……
……
……
……
…….

- ✓ **What are the tyoe of shell in UNIX operating system?**

……
……
……

……
…….

- ✓ **Discuss the about positional parameters**

……
……
……
……
…….

- ✓ **Discuss about variable in shell programming**

……
……
……
……
…….

8.19 Further Reading:

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/

Unit-9 : Decision Control Structures

Structure:

9.1 Introduction:

Conditional statements, also known as decision control structures, such as if, if-else, and switch, are crucial for decision-making in programming. These Decision-Making Statements evaluate one or more conditions to determine whether a specific block of code should be executed, thereby influencing the flow of program execution.

In everyday life, we make decisions that dictate our next actions. Similarly, in programming, conditional statements guide the flow of execution based on certain conditions. For instance, in C, you might use if x to run y, or else to run z. When dealing with multiple conditions, the if-else if structure helps you execute different code blocks based on varying conditions, such as running p if x occurs, q if y occurs, and r otherwise.

In this unit, we will delve into decision-making within Unix shell scripting. When crafting a shell script, you may encounter situations where you need to select between different paths. Conditional statements in shell scripts will allow you to make informed decisions and carry out the correct actions based on those choices.

9.1a Objectives

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of Decision statements
- if, else, elif, test command, nested if-else, forms of if,

- use of logical operators,
- case control structure

9.2 Types

There are following types of decision-making statements :

- ✓ if Statement
- ✓ if-else Statement
- ✓ Nested if Statement
- ✓ if-else-if Ladder
- ✓ switch Statement
- ✓ Conditional Operator
    - Jump Statements:
    - break
    - continue
    - goto
    - return

9.2.1 if statement:

The if statement is the simplest form of a decision-making statement. It determines whether a particular block of code should be executed based on a given condition. If the condition evaluates to true, the code within the if block runs; if the condition is false, the code is skipped.

Syntax:

if(*condition*)

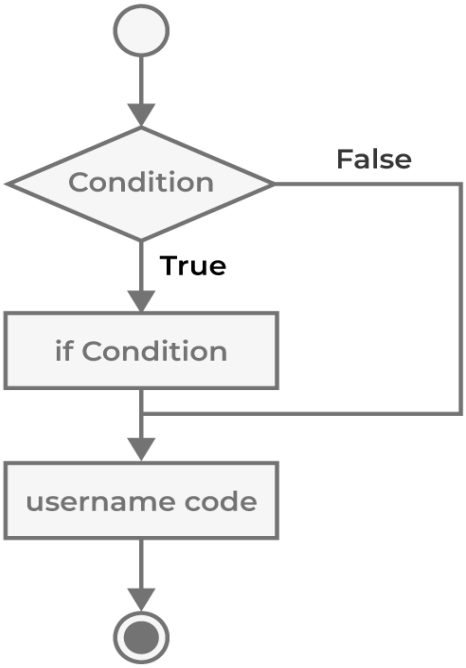
{

// Statements to execute if

// condition is true

}

The if statement is the most basic decision-making construct in programming. It checks a specified condition and decides whether to execute a particular block of code based on whether the condition is true. If the condition evaluates to true, the code inside the if block is executed; if the condition is false, the code is bypassed..



Example:

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
   int i = 10;

   if (i > 15) {
      printf("10 is greater than 15");
   }

   printf("I am Not in if");
}
```

Output

I am Not in if

9.2.2. if-else

The if statement alone handles the scenario where a block of code is executed only if a condition is true, and nothing happens if the condition is false. However, if you need to specify an alternative action when the condition is false, you can use the else statement in conjunction with the if statement.

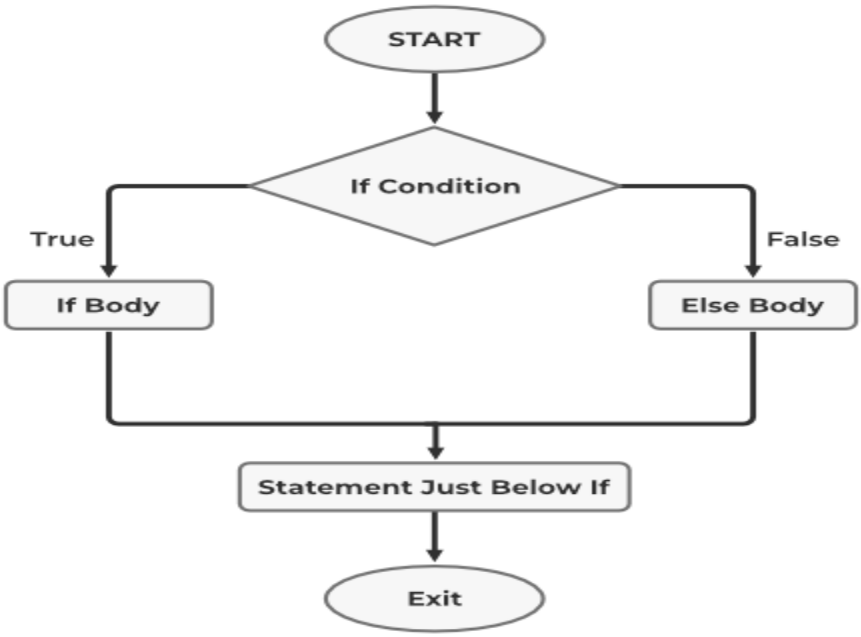
The if-else statement allows you to define two distinct blocks of code:

- One block executes if the condition evaluates to true.
- The other block executes if the condition evaluates to false.

Syntax of if else

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
   // condition is false
}
```

Flowchart of if-else



Example of if-else

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
   int i = 20;

   if (i < 15) {

      printf("i is smaller than 15");
   }
   else {

      printf("i is greater than 15");
   }
   return 0;
}
```

Output

i is greater than 15

9.2.3 if-else-if /nested if else:

The if-else if statements are employed when you need to choose among multiple options based on different conditions. The conditions are evaluated sequentially from top to bottom. As soon as one of the conditions evaluates to true, the corresponding block of code is executed, and the remaining conditions are skipped. If none of the conditions are met, the else block (if present) will be executed.

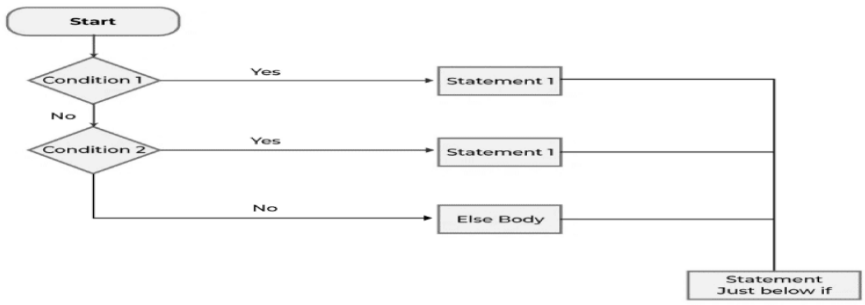
Syntax :

```
if (condition)
   statement;
else if (condition)
   statement;
.
.
```

else

statement;

Flowchart



Example

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

Output

i is 20

9.2.4 Switch Statement:

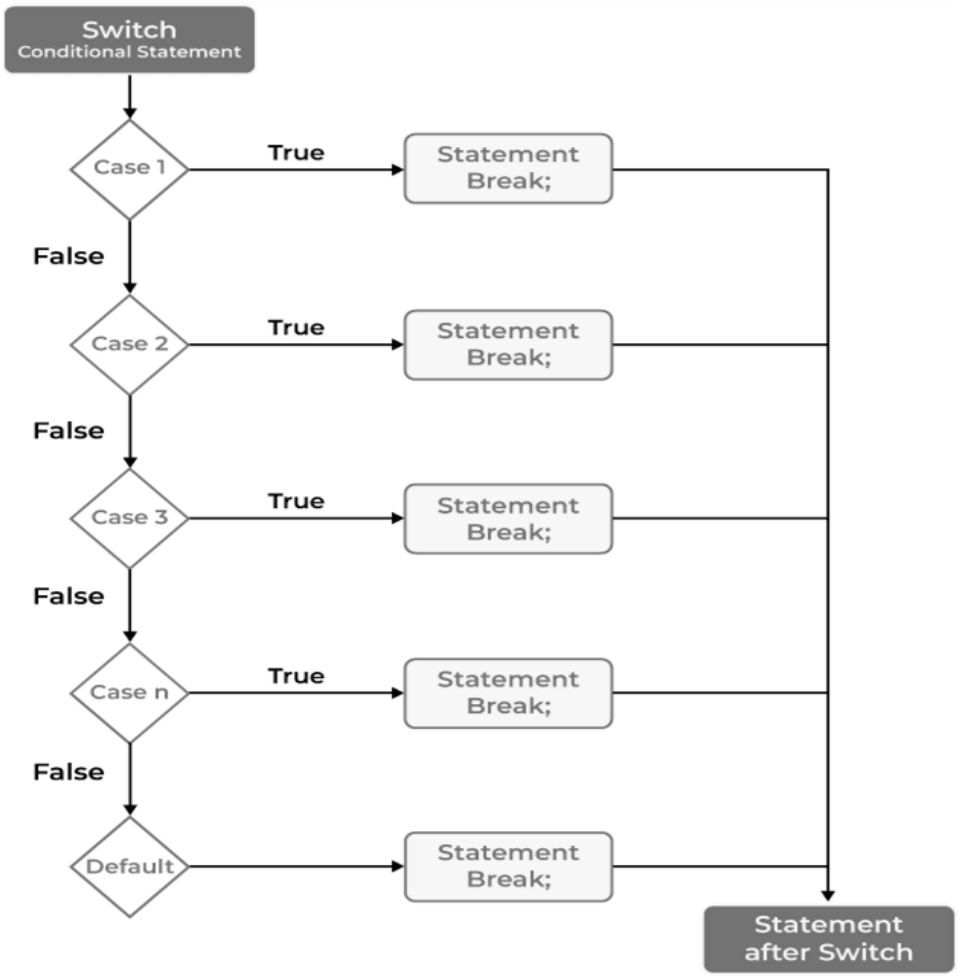
The switch statement is an alternative to the if-else if ladder, designed to execute different blocks of code based on the value of a variable. It provides a more streamlined way to handle multiple conditions that depend on the value of a single variable..

Syntax of switch

```
switch (expression) {
   case value1:
      statements;
   case value2:
      statements;
   ....
   ....
   ....
   default:
      statements;
}
```

Note: The switch expression should evaluate to either integer or character. It cannot evaluate any other data type.

Flowchart of switch



Example

```
// C Program to illustrate the use of switch statement
#include <stdio.h>

int main()
{
   // variable to be used in switch statement
   int var = 2;

   // declaring switch cases
   switch (var) {
   case 1:
      printf("Case 1 is executed");
      break;
   case 2:
      printf("Case 2 is executed");
      break;
   default:
      printf("Default Case is executed");
      break;
   }

   return 0;
}
```

Output

Case 2 is executed

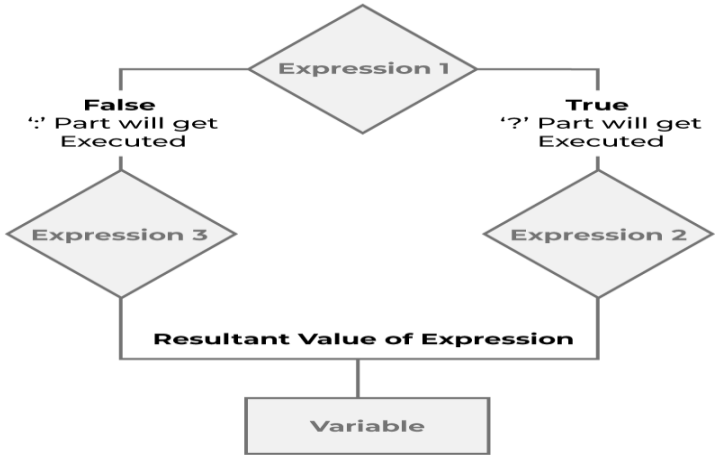
9.2.5 Conditional Operator

The conditional operator, often referred to as the ternary operator, provides a shorthand way to include conditional logic in your code. It functions similarly to an if-else statement but is more concise. It operates on three operands, which is why it's sometimes called the ternary operator.

Syntax

(*condition*) **?** *[true_statements]* **:** *[false_statements]*;

Flowchart of Conditional Operator



Example

```
// C Program to illustrate the use of conditional operator
#include <stdio.h>

// driver code
int main()
{

    int var;
    int flag = 0;

    // using conditional operator to assign the value to var
    // according to the value of flag
    var = flag == 0 ? 25 : -25;
    printf("Value of var when flag is 0: %d\n", var);

    // changing the value of flag
    flag = 1;
    // again assigning the value to var using same statement
    var = flag == 0 ? 25 : -25;
    printf("Value of var when flag is NOT 0: %d", var);
```

```
    return 0;
}
```

Output

Value of var when flag is 0: 25

Value of var when flag is NOT 0: -25

9.2.6 Jump Statements

In programming, jump statements are used to alter the flow of control in a program, allowing you to jump to different parts of the code. They are often used to break out of loops, skip code, or transfer control to specific locations in a function. Here are the four main types of jump statements:

A) Break

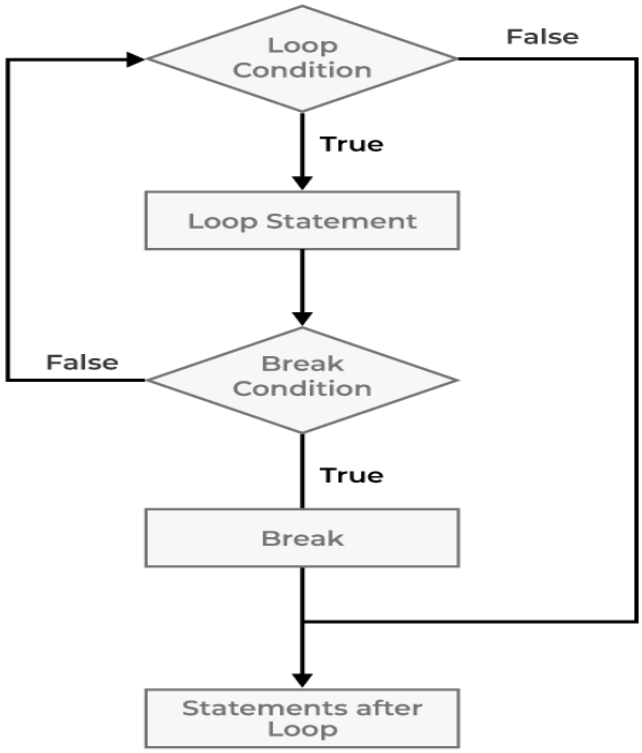
Purpose: Exits from the nearest enclosing loop (for, while, do-while) or switch statement.

Usage: Often used to terminate a loop early or exit from a switch case.

Syntax

```
break;
```

Flowchart:



Example of break

```
// C program to illustrate
// to show usage of break
// statement
#include <stdio.h>

void findElement(int arr[], int size, int key)
{
   // loop to traverse array and search for key
   for (int i = 0; i < size; i++) {
      if (arr[i] == key) {
         printf("Element found at position: %d",
               (i + 1));
         break;
      }
   }
}

int main()
{
   int arr[] = { 1, 2, 3, 4, 5, 6 };

   // no of elements
   int n = 6;

   // key to be searched
   int key = 3;

   // Calling function to find the key
   findElement(arr, n, key);

   return 0;
}
```

Output

Element found at position: 3

continue

Purpose: Skips the rest of the code inside the current iteration of a loop and proceeds with the next iteration.

Usage: Useful for skipping certain conditions within loops without exiting the loop entirely.

Syntax of continue

continue;

Example

```
// C program to explain the use
// of continue statement
#include <stdio.h>

int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            printf("%d ", i);
    }

    return 0;
}
```

Output

1 2 3 4 5 7 8 9 10

After if block 0

C) goto:

Purpose: Transfers control to a specified label within the same function.

Usage: Generally discouraged due to its potential to create confusing and hard-to-maintain code. It can, however, be used for jumping to specific parts of the code for error handling or exiting from deeply nested loops..

Syntax of goto

Syntax1 | **Syntax2**

---

goto *label;* | *label:*

. | .

. | .

. | .

label: | **goto** *label;*

Example

```
// C program to print numbers
// from 1 to 10 using goto
// statement
#include <stdio.h>

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
```

```
int main()
{
   printNumbers();
   return 0;
}
```

Output

1 2 3 4 5 6 7 8 9 10

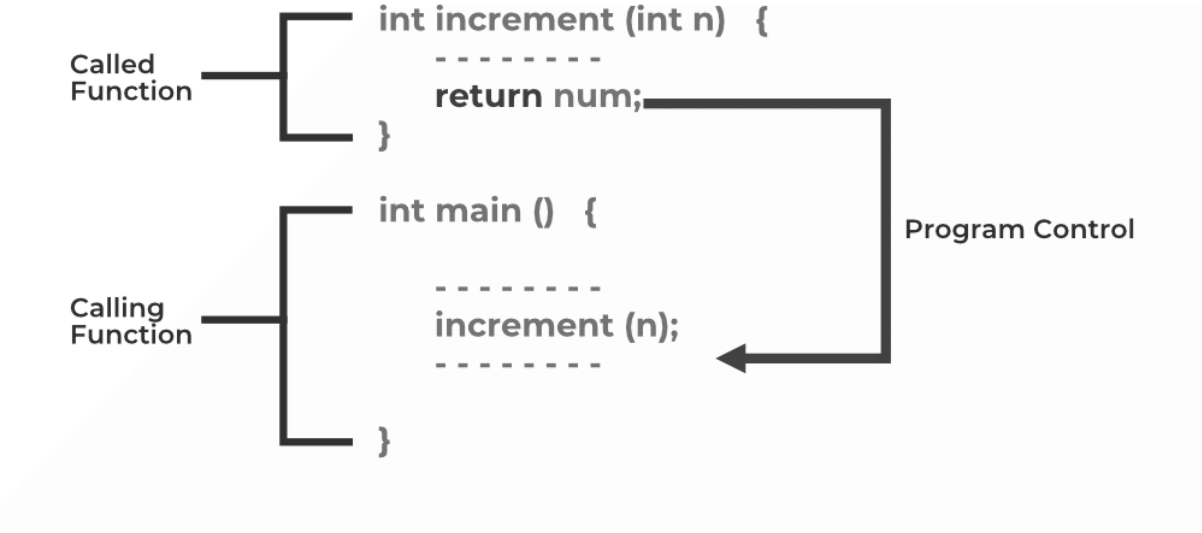
D) return

return **Statement**

- **Purpose**: Exits from the current function and optionally returns a value to the caller.
- **Usage**: Ends the execution of a function and can be used to return a result to the caller.

.

Flowchart of return



Syntax of return

return *[expression]*;

Example of return

C

// C code to illustrate return

// statement

#include <stdio.h>

// non-void return type

```
// function to calculate sum
int SUM(int a, int b)
{
   int s1 = a + b;
   return s1;
}

// returns void
// function to print
void Print(int s2)
{
   printf("The sum is %d", s2);
   return;
}

int main()
{
   int num1 = 10;
   int num2 = 10;
   int sum_of = SUM(num1, num2);
   Print(sum_of);
   return 0;
}
```

Output

The sum is 20

9.3 Form of if

In Unix Shell scripting, conditional statements allow you to execute different actions based on various conditions, similar to other programming languages. The primary forms of if...else statements in Unix Shell scripting are:

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

The if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

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

The *Shell expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

```
#!/bin/sh

a=10
b=20

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

output:

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

The if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]
then
```

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

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

```
#!/bin/sh

a=10
b=20

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

Upon execution, you will receive the following result −

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

The if...elif...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
  Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
  Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
```

then

Statement(s) to be executed if expression 3 is true

else

Statement(s) to be executed if no expression is true

fi

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

Live Demo

```
#!/bin/sh

a=10
b=20

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

Upon execution, you will receive the following result −

a is less than b

9.4 The case...esac Statement

In Unix Shell scripting, the case...esac statement is a powerful control flow structure used to handle multiple branches based on the value of a single variable. It serves a similar purpose to the switch...case statement found in other programming languages like C, C++, and Perl, providing a more efficient way to manage multiple conditions compared to using a series of if...elif statements.

The case...esac statement allows you to match a variable against several patterns and execute corresponding code blocks based on which pattern matches. It simplifies the code and makes it more readable when dealing with multiple conditions based on a single variable.

- **case...esac**: Efficiently handles multiple conditions based on a single variable, similar to switch...case in other languages.
- **Patterns**: Allow matching against specific values or using wildcards for broader matching.
- **Default Case (*)**: Provides a fallback option when no patterns match.

Using case...esac can make your shell scripts cleaner and more maintainable when you need to perform different actions based on the value of a single variable.

Syntax

```
case variable in
  pattern1)
    # Commands to execute if variable matches pattern1
    ;;
  pattern2)
    # Commands to execute if variable matches pattern2
    ;;
  ...
  *)
    # Commands to execute if no patterns match (default case)
    ;;
esac
```

Key Points

- **variable**: The variable whose value is being checked.
- **pattern1, pattern2, ...**: The patterns to match against the value of the variable. Patterns can include wildcards like * and ?.
- ***)**: The default case, executed if none of the specified patterns match.
- **;;**: Ends each case block.
- **esac**: Terminates the case statement (reverse of case).

Example

```
#!/bin/bash

day="Monday"

case $day in
```

```
    Monday)
        echo "It's Monday."
        ;;
    Tuesday)
        echo "It's Tuesday."
        ;;
    Wednesday)
        echo "It's Wednesday."
        ;;
    *)
        echo "It's another day."
        ;;
esac
```

In this example:

- If `day` is "Monday", it prints "It's Monday."
- If `day` is "Tuesday", it prints "It's Tuesday."
- If `day` is "Wednesday", it prints "It's Wednesday."
- For any other value of `day`, it prints "It's another day."

Example with Wildcards

```
#!/bin/bash

file="example.txt"

case $file in
    *.txt)
        echo "This is a text file."
        ;;
    *.jpg | *.png)
        echo "This is an image file."
        ;;
    *.sh)
        echo "This is a shell script."
        ;;
    *)
```

```
    echo "Unknown file type."
    ;;
esac
```

In this example:

- If file ends with .txt, it prints "This is a text file."
- If file ends with .jpg or .png, it prints "This is an image file."
- If file ends with .sh, it prints "This is a shell script."
- For any other file type, it prints "Unknown file type."

9.5 Test Statement

The test statement and its variations provide a robust way to handle conditional logic in Unix Shell scripting, enabling you to write flexible and effective scripts.

In Unix Shell scripting, the test statement (or its synonym [...]) is used to evaluate expressions and perform conditional checks. It's commonly used in if statements to determine whether certain conditions are met, allowing the script to make decisions based on those conditions.

Syntax

Using test

```
test expression
```

Using [...]

```
[ expression ]
```

Both test and [...] are functionally equivalent, but [...] is more commonly used due to its readability.

Types of Expressions

1. **File Tests**
    - **Check if a file exists**:

      ```
      if [ -e filename ]; then
          echo "File exists."
      fi
      ```

    - **Check if a file is readable**:

      ```
      if [ -r filename ]; then
          echo "File is readable."
      fi
      ```

    - **Check if a file is writable**:

      ```
      if [ -w filename ]; then
          echo "File is writable."
      fi
      ```

- **Check if a file is executable**:

```
if [ -x filename ]; then
   echo "File is executable."
fi
```

2. **String Tests**
   - **Check if a string is empty**:

```
if [ -z "$string" ]; then
   echo "String is empty."
fi
```

   - **Check if a string is not empty**:

```
if [ -n "$string" ]; then
   echo "String is not empty."
fi
```

   - **Check if two strings are equal**:

```
if [ "$string1" = "$string2" ]; then
   echo "Strings are equal."
fi
```

3. **Integer Tests**
   - **Check if an integer is greater than another**:

```
if [ $number1 -gt $number2 ]; then
   echo "$number1 is greater than $number2."
fi
```

   - **Check if an integer is less than another**:

```
if [ $number1 -lt $number2 ]; then
   echo "$number1 is less than $number2."
fi
```

These operators are used to test a particular property of a file.

- **-b operator**: This operator checks whether a file is a block special file or not. It returns true if the file is a block special file otherwise returns false.
- **-c operator**: This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise returns false.
- **-d operator**: This operator checks if the given directory exists or not. If it exists then operators returns true otherwise returns false.

- **-e operator**: This operator checks whether the given file exists or not. If it exists this operator returns true otherwise returns false.
- **-r operator**: This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise returns false.
- **-w operator**: This operator checks whether the given file has write access or not. If it has write then it returns true otherwise returns false.
- **-x operator**: This operator checks whether the given file has execute access or not. If it has execute access then it returns true otherwise returns false.
- **-s operator**: This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it returns false.

9.6 Logical Operators in Shell Scripting

In programming, decisions and flow control are managed using logical operators, also known as Boolean operators. These operators are used to combine multiple conditions and determine the overall outcome based on these conditions. This process is often referred to as flow control..

Logical OR (||) in Shell Scripting

The logical OR operator (||) is used to combine two or more conditions, where the overall expression evaluates to true if at least one of the conditions is true. This operator helps in scenarios where you need to check multiple conditions and want the code block to execute if any of them are met..

The given below scripts shows the use of Logical OR (||):

```
#!/bin/bash

read -p "Try your Guess: " num

if [[ $num%2==0 || $num%5==0 ]]  # if num is divisible by 2 or 5, it is right guess
then
  echo "You guess correctly."
else
  echo "Better Luck Next Time."
fi
```

Input

10

Output

You guess correctly.

Logical AND (&&) in Shell Scripting

The logical AND operator (&&) is used to combine multiple conditions in such a way that the overall expression evaluates to true only if all the individual conditions are true. In shell scripting, this operator ensures that subsequent conditions are only evaluated if the preceding conditions are true, utilizing short-circuit evaluation.

```
!/bin/bash
read -p n
if [ $((n % 2)) == 0 ] && [ $((n % 5)) == 0 ];
then
   echo "Number is even and also divisible by 5"
else
   echo "Number is odd and not divisible by 5"
fi
```

Input

10

Output

Number is even and also divisible by 5

Multiple Logical OR & AND in Shell Scripting

We can use multiple Logical OR & AND operators in one single-line statement. The given below scripts show the use of multiple logical operators in one statement:

```
#!/bin/bash

# Check if the number is negative or positive
# Or if the number is positive, it lies within 50-60 or not
 read -p "Please enter a number: " n
 if ( [ $n -le 0 ]) || ([ $n -ge 50 ] && [ $n -le 60 ])
then
    echo "Number is negative or between 50 and 60"
else
    echo "Input number ($n) didn’t exist"
fi
```

Input

40

Output

Input number (40) didn’t exist

9.7 SUMMARY

In this unit, we have learn about the basic concept of Decision control statements like if, else, elif, test command, nested if-else, forms of if, use of logical operators, case control structure in shell programming

9.8 Check your progress:

- **What do you mean by Decision statements ?**

……
……
……
……

- ✓ **What are the forms of if in shell?**

……
……
……
……

- ✓ **Discuss the about if else statement in shell script.**

……
……
……
……

- ✓ **Discuss about case control structure in shell programming**

……
……
……
……

Further Reading:

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/

Unit-10 : Loop control structures

Structure

10.1 Introduction

In Unix Shell scripting, loops are essential for automating repetitive tasks and executing commands multiple times based on certain conditions. By using loops, you can simplify your scripts and avoid writing repetitive code. The main types of loops in shell scripting are for, while, and until. Each loop type serves different purposes and can be controlled using various keywords. Loops are fundamental in shell scripting for automating repetitive tasks and managing iterative processes efficiently.

10.1a Objectives

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of looping statements
- while, until, for,
- nested loops,
- break statement,
- continue statement.

10.2 Types of Loops in Shell Scripting

- ✓ **for Loop**: Iterates over a list of items or a range of values.
- ✓ **while Loop**: Executes a block of code as long as a condition is true.
- ✓ **until Loop**: Executes a block of code as long as a condition is false.
- ✓ **Controlling loop**: **break** to exit a loop and **continue** to skip to the next iteration.

10.2.1 for *Loop*

The for loop iterates over a list of items, executing a block of code for each item. It's useful for iterating over a range of values or a list of items. In shell scripting, the for loop is a versatile construct that allows you to execute a set of commands multiple times, either iterating over a list of items or a range of values. It simplifies the process of performing repetitive tasks by automating iterations based on specified criteria. The for loop in shell scripting is a powerful tool for executing commands multiple times. By iterating over lists, ranges, command outputs, or command-line arguments, you can automate repetitive tasks efficiently. The flexibility of the for loop helps you manage different types of iterations and tailor your scripts to meet various needs.

- **Syntax**:

  for variable in list

  do

  # Commands to execute

  done

- **Example**:

  # Loop through a list of numbers

  for i in 1 2 3 4 5

  do

  echo "Number: $i"

  done

  In this example, the loop iterates over the numbers 1 through 5, printing each number.

- **Example with Range**:

  # Loop through a range of numbers

  for i in {1..5}

  do

  echo "Number: $i"

  done

  This example uses brace expansion to generate a sequence of numbers from 1 to 5.

10.2.2 While Loop

In shell scripting, the while loop is used to repeatedly execute a block of commands as long as a specified condition evaluates to true. The loop continues running until the condition becomes false. The while loop is particularly useful when you want to perform operations as long as certain criteria are met, and it provides a way to handle situations where the number of iterations isn't known in advance.

The while loop is a fundamental control structure in shell scripting used to perform repetitive tasks based on a condition. It starts by evaluating the condition, and if true, it executes the commands within the loop block. The loop continues to run until the condition evaluates to false, allowing you to automate tasks efficiently when the number of iterations is not predetermined.

Key Points

- **Condition**: The loop's execution depends on the condition provided. It can involve comparisons, string checks, or command outputs.
- **Infinite Loop**: If the condition is always true, the while loop will run indefinitely. To prevent this, ensure the condition changes in a way that will eventually become false.

How It Works

1. **Initial Condition Evaluation**: Before the loop starts, the condition is evaluated.
2. **Execution**: If the condition is true, the commands inside the loop are executed.
3. **Re-evaluation**: After executing the commands, the condition is evaluated again.
4. **Termination**: The loop continues as long as the condition remains true. If the condition becomes false, the loop exits and the script continues with the commands following the done keyword.

Syntax:

```
while [ condition ]
do
   # Commands to execute
Done
```

Explanation

- ✓ **while**: The keyword that begins the loop.
- ✓ **[condition]**: The condition to be evaluated. If it evaluates to true (non-zero exit status), the commands within the loop are executed.
- ✓ **do**: Marks the start of the block of commands that will be executed if the condition is true.
- ✓ **done**: Marks the end of the while loop block.

Example:

```
# Initialize a variable
count=1

# Loop while the condition is true
while [ $count -le 5 ]
do
   echo "Count: $count"
   count=$((count + 1))
done
```

This example prints numbers from 1 to 5. The loop continues as long as $count is less than or equal to 5.

10.2.3 until Loop

The until loop is similar to the while loop but executes the block of code as long as the condition is false. The loop exits when the condition becomes true. The until loop in shell scripting is indeed similar to the while loop but operates with the opposite logic. While the while loop executes its body as long as a condition is true, the until loop continues to execute its body until the condition becomes true. This distinction makes the until loop useful for situations where you need to perform tasks until a specific condition is met.

How It Works

1. **Initial Condition Evaluation**: The condition is evaluated before entering the loop.
2. **Execution**: If the condition is false, the commands within the loop are executed.
3. **Re-evaluation**: After executing the commands, the condition is evaluated again.
4. **Termination**: The loop continues as long as the condition remains false. When the condition becomes true, the loop exits, and execution continues with the commands following the done keyword.

- **Syntax**:

  ```
  until [ condition ]
  do
     # Commands to execute
  done
  ```

- **Example**:

  ```
  # Initialize a variable
  count=1

  # Loop until the condition is true
  until [ $count -gt 5 ]
  do
     echo "Count: $count"
     count=$((count + 1))
  done
  ```

In this example:

- The count variable starts at 1.
- The until loop runs as long as count is less than or equal to 5 (i.e., until the condition [$count -gt 5] becomes true).

- Each iteration prints the current value of count and then increments it.
- The loop exits when count becomes greater than 5.

Controlling Loops

You can control loop execution using specific keywords:

- **break**: Exits the loop immediately.
    - **Example**:

      ```
      for i in {1..10}
      do
        if [ $i -eq 5 ]; then
          break
        fi
        echo "Number: $i"
      done
      ```

      This loop prints numbers from 1 to 4 and exits when `$i` equals 5.
- **continue**:

  Skips the current iteration and continues with the next iteration of the loop.
    - **Example**:

      ```
      for i in {1..10}
      do
        if [ $i -eq 5 ]; then
          continue
        fi
        echo "Number: $i"
      done
      ```

      This loop prints numbers from 1 to 10, skipping the number 5.

10.3 Nested Loops

In shell scripting, nested loops refer to the use of one loop inside another. This technique allows you to perform more complex iterations and handle multi-dimensional data or scenarios requiring multiple layers of repetition. Nested loops can be particularly useful when working with matrices, tables, or any situation where you need to iterate over two or more dimensions.Nested loops in shell scripting allow for iterating through multi-dimensional data by embedding one loop inside another. You can use any combination of loop types (for, while, until) to achieve the desired iterations. Be mindful of the increased complexity and potential performance impacts when using nested loops, and strive to keep your scripts as clear and efficient as possible

When using nested loops in shell scripting, several key considerations can help ensure that your scripts are efficient, maintainable, and clear. Here are the main aspects to keep in mind:

1. Complexity Management
    - **Understandability**: Nested loops can become complex quickly, making your script harder to understand. Ensure that the logic is well thought out and easy to follow.
    - **Simplicity**: Try to simplify the logic where possible. Complex nested loops might be a sign that you need to rethink your approach or break the problem into smaller, more manageable pieces.
2. Performance
    - **Execution Time**: Nested loops can significantly impact performance, especially if the number of iterations grows exponentially. For example, a loop inside a loop that both iterate over large ranges will result in a quadratic number of iterations.
    - **Optimization**: Evaluate if there are more efficient algorithms or data structures that can replace nested loops. For instance, use indexing or hash tables to reduce the need for deep nesting.
3. Readability and Maintainability
    - **Indentation**: Properly indent nested loops to make the script more readable. Each level of nesting should be clearly visible.
    - **Comments**: Add comments to explain the purpose of each loop and how they interact. This helps others (and yourself) understand the script later.
    - **Descriptive Variable Names**: Use meaningful variable names for loop counters and iterates to make the purpose of each loop clearer.
4. Avoid Infinite Loops
    - **Termination Conditions**: Ensure that the inner and outer loops have proper termination conditions to avoid infinite loops. An infinite loop can freeze or crash your script.
    - **Testing**: Test the script with various inputs to ensure that loops terminate as expected.
5. **Resource Management**
    - **Memory Usage**: Be aware of the memory usage implications of deeply nested loops. For instance, if each loop iteration creates large objects or performs intensive calculations, it can consume a lot of memory.
    - **Resource Leaks**: Ensure that any resources (like file handles or network connections) are properly managed within nested loops. Failure to close resources can lead to leaks and system performance issues.
6. **Debugging**
    - **Tracing**: Use debugging techniques like `echo` statements or a debugger to trace through nested loops. This can help identify where things might be going wrong.
    - **Simplify for Testing**: Simplify the loops or use smaller data sets during testing to make debugging easier.
7. **Loop Control**
    - **`break` and `continue`**: Utilize `break` and `continue` statements effectively to control the flow of nested loops. `break` can exit the innermost loop, while `continue` can skip to the next iteration of the innermost loop.

- **Labeling**: In more complex scenarios, you might use labels (in languages that support them) to control specific nested loops, though this is less common in shell scripting.

Syntax for Nested Loops

The general syntax for nested loops in shell scripting involves placing one loop inside the body of another loop. The inner loop executes its commands for each iteration of the outer loop

Example

```
#!/bin/bash
for i in {1..3}
do
   echo "Outer loop iteration: $i"
   for j in {A..C}
   do
      echo "  Inner loop iteration: $j"
   done
done
```

Explanation:

- The outer for loop iterates over the values 1 through 3.
- For each iteration of the outer loop, the inner for loop iterates over the values A through C.
- This results in each combination of i and j being processed, with j running through its full range for each value of i.

Example with while Loops

```
#!/bin/bash

i=1

while [ $i -le 3 ]
do
   echo "Outer loop iteration: $i"
   j=1
   while [ $j -le 3 ]
   do
      echo "  Inner loop iteration: $j"
      j=$((j + 1))
   done
   i=$((i + 1))
```

```
done
```

Explanation:

- The outer `while` loop increments `i` from 1 to 3.
- For each value of `i`, the inner `while` loop increments `j` from 1 to 3.
- This produces a nested iteration where `j` runs its full range for each value of `i`.

Example with Mixed Loop Types

sh

Copy code

```
#!/bin/bash

for i in {1..2}
do
   echo "Outer loop iteration: $i"
   while [ $i -le 2 ]
   do
      echo "  Inner loop while iteration: $i"
      i=$((i + 1))
      if [ $i -gt 2 ]; then
         break
      fi
   done
done
```

Explanation:

- The outer `for` loop iterates over values 1 and 2.
- The inner `while` loop increments `i` and breaks if `i` exceeds 2.
- This demonstrates using mixed loop types and controlling loop execution with conditions and `break` statements.

10.4 Working with Multidimensional Arrays in Nested Loops

In Bash scripting, while true multidimensional arrays are not directly supported, you can simulate them using associative arrays or indexed arrays. This involves organizing data in a way that mimics a grid or table structure and then using nested loops to access and manipulate the data. Here's how you can achieve this with associative arrays and nested loops:

Simulating a Multidimensional Array

Using Associative Arrays

Associative arrays (or dictionaries) in Bash can be used to simulate multidimensional arrays. You can use a combination of keys to represent rows and columns.

Example: Simulating a 2D Grid

```
#!/bin/bash

# Declare associative arrays
declare -A grid

# Populate the grid
grid["1,1"]="A1"
grid["1,2"]="A2"
grid["1,3"]="A3"
grid["2,1"]="B1"
grid["2,2"]="B2"
grid["2,3"]="B3"
grid["3,1"]="C1"
grid["3,2"]="C2"
grid["3,3"]="C3"

# Loop through rows
for row in {1..3}
do
   echo "Row $row:"
   # Loop through columns
   for col in {1..3}
   do
      # Access the element in the grid
      echo -n "${grid[$row,$col]} "
   done
   echo # New line after each row
done
```

Explanation:

- **Associative Array Declaration**: `declare -A grid` initializes an associative array named `grid`.
- **Populating Data**: Data is stored with keys in the format `"row,column"`.
- **Nested Loops**: The outer loop iterates through rows, and the inner loop iterates through columns, accessing elements using the combined keys.

Using Indexed Arrays

For simpler cases, indexed arrays can also be used to simulate multidimensional arrays. This method uses an array of arrays where each sub-array represents a row.

Example: Simulating a 2D Grid

```
#!/bin/bash

# Declare indexed arrays for each row
row1=("A1" "A2" "A3")
row2=("B1" "B2" "B3")
row3=("C1" "C2" "C3")

# Loop through rows
for i in {0..2}
do
   echo "Row $((i+1)):"
   # Loop through columns
   for j in {0..2}
   do
      # Access the element in the grid
      echo -n "${!row$i[$j]} "
   done
   echo # New line after each row
done
```

Explanation:

- **D,erkcxcdcIndexed Arrays**: Arrays row1, row2, and row3 represent each row in the grid.
- **Nested Loops**: The outer loop iterates through rows, and the inner loop iterates through columns to access elements.

Key Considerations

When working with simulated multidimensional arrays in Bash, particularly using associative arrays or indexed arrays, there are several key considerations to keep in mind to ensure that your script remains efficient, maintainable, and error-free.

1. **Data Structure Choice**

- **Associative Arrays**: Ideal for scenarios where the indices or keys are non-numeric or when the grid size is dynamic. They provide flexibility with key-value pairs and can represent sparse matrices.

- **Indexed Arrays**: Suitable for fixed-size grids where you need a simple row-column structure. They are generally easier to use and understand when indices are numeric and consistent.

2. **Complexity and Readability**
   - **Clarity**: Ensure that your approach to simulating multidimensional arrays is clear and understandable. Properly comment your code to explain the structure and logic, especially if using complex key patterns or nested loops.
   - **Maintainability**: Choose a data structure that aligns with the complexity of your problem. Simpler structures are easier to maintain and debug.
3. **Performance**
   - **Efficiency**: Associative arrays can incur overhead due to hash table operations, making them potentially slower for large grids. Indexed arrays generally offer faster access times due to direct index-based access.
   - **Memory Usage**: Both types of arrays can consume significant memory if used with large datasets. Monitor memory usage to ensure your script performs well.
4. **Indexing and Key Management**
   - **Consistent Indexing**: When using indexed arrays, ensure that indices are consistently managed and within bounds. Out-of-bounds errors can cause runtime issues.
   - **Unique Keys**: With associative arrays, ensure that keys are unique and properly formatted. Duplicate keys or incorrect formatting can lead to data overwrite or access issues.
5. **Error Handling**
   - **Validation**: Validate inputs and conditions within loops to prevent unexpected behaviour. For example, check that keys exist in associative arrays before accessing them.
   - **Boundary Conditions**: Handle edge cases where the grid might have non-standard dimensions or where data might be missing.
6. **Loop Efficiency**
   - **Nested Loops**: Be cautious with deeply nested loops, as they can lead to increased execution time, especially for large datasets. Optimize loop logic where possible.
   - **Control Statements**: Use `break`, `continue`, or other control statements judiciously to manage loop flow and avoid unnecessary iterations.
7. **Data Access Patterns**
   - **Accessing Elements**: Ensure that your method for accessing elements (e.g., using combined keys or indices) is well-defined and efficient.
   - **Updating Data**: When updating values in the simulated grid, ensure that the correct elements are targeted and updated as intended.
8. **Testing and Debugging**
   - **Testing**: Test your script with various data sizes and structures to ensure it behaves as expected. Use smaller datasets initially to simplify debugging.
   - **Debugging**: Utilize debugging techniques such as `echo` statements or `set -x` to trace and identify issues within nested loops and array accesses.

10.5 Nested Loops for Automating Data

Automating data processing tasks using nested loops in Bash scripting can greatly enhance efficiency, especially when dealing with complex data structures like directories with multiple files. By leveraging nested loops, you can systematically and automatically handle tasks like renaming files, thus reducing manual intervention and minimizing human error.In Bash scripting, automation of data processing using nested loops can significantly streamline tasks that involve multiple layers of data. One practical example of this is renaming files in a directory structure where each subdirectory contains files with random names. By using nested loops, you can efficiently automate the renaming process across all directories and subdirectories.

Example: Renaming Files in a Directory Structure

Let's say you have a directory structure like this:

bash

Copy code

```
/parent_directory/
    /subdir1/
        file1.txt
        file2.txt
    /subdir2/
        fileA.txt
        fileB.txt
    /subdir3/
        fileX.txt
        fileY.txt
```

You want to rename all files in each subdirectory to a systematic naming convention, such as prefixing them with "new_". You can use nested loops in a Bash script to achieve this.

Bash Script to Rename Files

```
#!/bin/bash

# Define the parent directory
parent_dir="/path/to/parent_directory"

# Loop through each subdirectory in the parent directory
for subdir in "$parent_dir"/*/; do
    # Check if the subdir is indeed a directory
    if [ -d "$subdir" ]; then
        echo "Processing directory: $subdir"
```

```
    # Loop through each file in the current subdirectory
    for file in "$subdir"*; do
      # Check if the item is indeed a file
      if [ -f "$file" ]; then
        # Extract the base name and extension
        base_name=$(basename "$file")
        extension="${base_name##*.}"
        new_name="new_${base_name}"

        # Rename the file
        mv "$file" "$subdir$new_name"
        echo "Renamed $file to $subdir$new_name"
      fi
    done
  fi
done
```

Explanation

1. **Define Parent Directory**: Set the path to the parent directory where the subdirectories are located.
2. **Outer Loop**: Iterate through each subdirectory inside the parent directory.
    - `"$parent_dir"/*/` matches all directories within the parent directory.
    - `-d "$subdir"` checks if the item is a directory.
3. **Inner Loop**: Iterate through each file within the current subdirectory.
    - `"$subdir"*` matches all files within the current subdirectory.
    - `-f "$file"` checks if the item is a file.
4. **Rename File**: Use `mv` to rename the file.
    - Extract the base name of the file and the extension.
    - Construct the new file name with a prefix.
    - Rename the file and print a message confirming the action.

Other key term:

- **Path Variables**: Ensure that the `parent_dir` variable is correctly set to the path where your directories are located.
- **Permissions**: Make sure you have the necessary permissions to read from and write to the directories and files.
- **Testing**: Test the script with a small subset of files or in a controlled environment to ensure it works as expected before running it on a large number of files.

- **Error Handling**: Add error handling (e.g., checking if mv succeeds) to handle cases where files may not be renamed due to permission issues or other errors.

10.6 Reading from a file

Reading from a file in Bash scripting is a fundamental operation that allows you to process and manipulate data stored in text files. Reading from a file in Bash scripting can be done using various methods, depending on your needs:

- ✓ **cat**: Simple and straightforward for displaying file contents.
- ✓ **while loop with read**: Effective for line-by-line processing.
- ✓ **for loop with file expansion**: Useful for small files or when you need array processing.
- ✓ **awk and sed**: Powerful tools for text processing and manipulation.

1. **Using** cat **Command**

The cat command is a simple way to display the contents of a file. While it's often used to view the file, it can also be piped into other commands for processing.

Example: Display File Contents

#!/bin/bash

Display the contents of a file

cat /path/to/file.txt

Example: Pipe File Contents

#!/bin/bash

Process file contents with another command

cat /path/to/file.txt | grep "search_term"

2. **Using** while **Loop with** read

This method reads a file line by line, which is useful for processing each line individually. This approach is particularly useful when dealing with large files or when you need to perform operations on each line.

Example: Reading Line by Line

sh

Copy code

#!/bin/bash

Define the file to read

file="/path/to/file.txt"

```
# Read the file line by line
while IFS= read -r line; do
    echo "Processing: $line"
    # You can add more processing logic here
done < "$file"
```

Explanation:

- IFS= ensures that leading/trailing whitespace is preserved.
- -r prevents read from interpreting backslashes as escape characters.
- < "$file" redirects the file to the while loop.

3. **Using** for **Loop with File Expansion**

You can also use a for loop to iterate over lines if the file is small and you need to read each line into a variable.

Example: Using for Loop

```
#!/bin/bash

# Define the file to read
file="/path/to/file.txt"

# Read the file into an array
mapfile -t lines < "$file"

# Loop through each line in the array
for line in "${lines[@]}"; do
    echo "Processing: $line"
    # Add additional processing logic here
done
```

Explanation:

- mapfile -t (or readarray -t in some versions) reads the file into an array, with each line as an element.
- ${lines[@]} expands to each line in the array.

4. **Using** awk

The awk command is a powerful tool for text processing and can be used to read and process file content in a variety of ways.

Example: Print Specific Fields

sh

Copy code

```
#!/bin/bash

# Process the file with awk
awk '{ print "Field 1: " $1 ", Field 2: " $2 }' /path/to/file.txt
```

Explanation:

- `$1`, `$2`, etc., represent fields in each line of the file. `awk` splits lines into fields based on whitespace by default.

5. **Using** sed

The `sed` command is useful for text transformations and can be used to read and modify file contents.

Example: Replace Text

sh

Copy code

```
#!/bin/bash

# Replace 'old_text' with 'new_text' in the file
sed 's/old_text/new_text/g' /path/to/file.txt
```

Explanation:

- `s/old_text/new_text/g` performs a global search-and-replace in the file.

6. **Handling Special Characters**

When reading files with special characters, such as spaces or newline characters, make sure to handle them properly to avoid issues.

Example: Handling Special Characters

sh

Copy code

```
#!/bin/bash

# Read the file line by line, handling special characters
while IFS= read -r line; do
    echo "Processing line: $line"
    # Ensure special characters are handled correctly
done < "/path/to/file_with_special_chars.txt"
```

10.7 Using command line arguments

Using command line arguments in Bash scripts enhances their flexibility and usability. You can:

- Access individual arguments using $1, $2, etc.

- Use $# to get the number of arguments.
- Utilize $@ to handle all arguments as a list.

By incorporating these techniques, you can create more versatile and user-friendly scripts that can handle a variety of tasks based on user input.

Using command line arguments in Bash scripts allows you to make your scripts more dynamic and flexible by allowing users to pass inputs directly from the command line. This is especially useful for tasks such as processing files where the file names or paths can be specified by the user.

Accessing Command Line Arguments

In Bash, command line arguments are accessed using special variables:

- $1, $2, $3, etc.: Represent the first, second, third, etc., arguments.
- $#: Represents the total number of arguments.
- $@: Represents all arguments as a list.
- $*: Represents all arguments as a single string.

Examples of Using Command Line Arguments

Example 1:

Suppose you want to create a script that takes a filename as an argument and prints the contents of that file.

Script: print_file.sh

sh

Copy code

```
#!/bin/bash

# Check if exactly one argument is provided
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

# Assign the argument to a variable
filename="$1"

# Check if the file exists
if [ ! -f "$filename" ]; then
    echo "File not found: $filename"
    exit 1
fi
```

```
# Print the contents of the file
cat "$filename"
```

Usage:

```
bash print_file.sh /path/to/file.txt
```

Explanation:

- The script checks if exactly one argument is provided.
- It verifies if the specified file exists.
- It prints the file contents using `cat`.

Example 2: Line-by-Line Processing with Arguments

Create a script that processes a file line by line, using command line arguments to specify the file.

Script: process_lines.sh

```
sh
Copy code
#!/bin/bash

# Check if exactly one argument is provided
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

# Assign the argument to a variable
filename="$1"

# Read and process each line in the file
while IFS= read -r line; do
    echo "Processing line: $line"
    # Add additional processing logic here
done < "$filename"
```

Usage:

```
bash process_lines.sh /path/to/file.txt
```

Explanation:

- The script reads the filename from the command line argument.

- It processes each line of the file, printing a message for each line.

Example 3: Multiple Files Processing

Create a script that processes multiple files specified as command line arguments.

Script: process_multiple_files.sh

```
#!/bin/bash
# Check if at least one argument is provided
if [ "$#" -lt 1 ]; then
   echo "Usage: $0 <file1> [file2 ... fileN]"
   exit 1
fi
# Loop through each command line argument
for filename in "$@"; do
   # Check if the file exists
   if [ -f "$filename" ]; then
      echo "Processing file: $filename"
      # Add file processing logic here
      cat "$filename"  # Example: Print file contents
   else
      echo "File not found: $filename"
   fi
done
```

Usage:

bash process_multiple_files.sh /path/to/file1.txt /path/to/file2.txt

Explanation:

- The script loops through all command line arguments ($@).
- It processes each file if it exists, otherwise prints a "File not found" message.

10.8 SUMMARY: In this unit, we have learn about the basic concept of looping statements: while, until, for, reading from a file, using with command line arguments, nested loops, break statement, continue statement

10.9 Check your progress

- **What do you mean looping in shell script?**

……
……
……
……
…….

- ✓ **What are the tyoe of loop in shell ?**

……
……
……
……
…….

- ✓ **Discuss the about break statement**

……
……
……
……
…….

- ✓ **Discuss about nested loop in shell programming**

……
……
……
……
…….

10.10 Further Reading:

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/

BLOCK-4 INTRODUCTION

Block-4 is a course on "**Shell scripting in Unix**". This block course contains four units. Unit 11 is an introductory unit on **sed editor**. With this unit, the learners will be acquainted with some important concepts SED Editor like overview, uses of sed, sed operation, standard operations, pattern addressing, regular expressions, line information, I/O processing, yanking, putting, branching commands, multiline input processing. Unit 12 discusses about the **Bash scripting**. This includes the Bash scripting:Variables- variable assignment and variable scope, Operators, Command Line Arguments, Setting Values of Positional Parameters, Using Shift on Positional Parameters. Unit 13 concentrates on Control Flow in **Bash scripting**. In this unit, the learners will be acquainted with some important Control Flow Statements-Decision, loops and case statements, Arithmetic in Shell Script, Array, File and String Tests. Unit 14 is the last unit and it discusses about **gawk programming**.and cover the topic lke overview, command line syntax, standard options, Built in variables, operators, variable and array assignment, escape sequences, patterns and procedures, functions, file inclusion, output redirections, printf formats.

Learners will explore Shell scripting in Unix, including the sed editor, Bash scripting, and gawk programming. Each unit features boxes to clarify challenging terms and "ACTIVITY" sections for practical application. Relevant concepts are also highlighted in the "LET US KNOW" sections alongside the text.. And, at the end of each section, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be better if you solve the given problems in these boxes immediately, after you finish reading the section in which these questions occur and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

Unit-11 Sed editor

Structure

11.1 Introduction

Sed, short for **"stream editor**," is a robust command-line utility available on Unix/Linux systems. It enables users to perform straightforward text modifications on an input stream, which can be a file or data from a pipeline. Sed is particularly effective for applying bulk changes across numerous files or for editing files that are challenging to manage with a traditional text editor. **Sed** is a versatile tool for manipulating text streams, offering a broad array of features for searching, matching, and replacing text. This makes it well-suited for a variety of tasks and use cases involving text processing.

Syntax:

The basic syntax for the sed command is:

$ sed 'command' file

Here, the command is enclosed in single quotes, which is crucial for sed to interpret it correctly. The file is the input file that sed will process. If you're using sed with input from a pipeline, you don't need to specify a file argument.

11.1a.Objectives

After studying this unit ,the learner would be able to know about the following:-

- Basic of SED (**Stream editor)**
- What are the uses of sed,

- Basic sed operation
- Understanding of pattern addressing in sed
- regular expressions and its basic component of sed
- line information
- I/O processing
- yanking
- putting
- branching commands
- multiline input processing.

11.2 SED environment

Installation Using Package Manager

"Generally, SED is available by default on most GNU/Linux/unix distributions. Use **which** command to identify whether it is present on your system or not. If not, then install SED on Debian based GNU/Linux using **apt** package manager as follows:

[jery]$ sudo apt-get install sed

After installation, ensure that SED is accessible via command line.

[jerry]$ sed --version

On executing the above code, you get the following result:

sed (GNU sed) 4.2.2

Copyright (C) 2012 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later .

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.

Written by Jay Fenlason, Tom Lord, Ken Pizzini,

and Paolo Bonzini.

GNU sed home page: .

General help using GNU software: .

E-mail bug reports to: .

Be sure to include the word "sed" somewhere in the "Subject:" field.

Similarly, to install SED on RPM based GNU/Linux, use yum package manager as follows:

[root]# yum -y install sed

After installation, ensure that SED is accessible via command line.

[root]# sed --version

On executing the above code, you get the following result:

GNU sed version 4.2.1

Copyright (C) 2009 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE,

to the extent permitted by law.

GNU sed home page: .

General help using GNU software: .

E-mail bug reports to: .

Be sure to include the word "sed" somewhere in the "Subject:" field.

Installation from Source Code

As GNU SED is a part of the GNU project, its source code is available for free download. We have already seen how to install SED using package manager. Let us now understand how to install SED from its source code.

The following installation is applicable to any GNU/Linux software, and for most other freely-available programs as well. Here are the installation steps:

- Download the source code from an authentic place. The command-line utility **wget** serves this purpose.

  [jerry]$ wget ftp://ftp.gnu.org/gnu/sed/sed-4.2.2.tar.bz2
- Decompress and extract the downloaded source code.

  [jerry]$ tar xvf sed-4.2.2.tar.bz2
- Change into the directory and run configure.

  [jerry]$ ./configure
- Upon successful completion, the **configure** generates Makefile. To compile the source code, issue a **make** command.

  [jerry]$ make
- You can run the test suite to ensure the build is clean. This is an optional step.

  [jerry]$ make check
- Finally, install the SED utility. Make sure you have superuser privileges.

  [jerry]$ sudo make install

That is it! You have successfully compiled and installed SED. Verify it by executing

the **sed** command as follows:

[jerry]$ sed --version

On executing the above code, you get the following result:

sed (GNU sed) 4.2.2

Copyright (C) 2012 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later .

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.

Written by Jay Fenlason, Tom Lord, Ken Pizzini,

and Paolo Bonzini.

GNU sed home page: .

General help using GNU software: .

E-mail bug reports to: .

Be sure to include the word "sed" somewhere in the "Subject:" field.”

Resources link: https://www.tutorialspoint.com/sed/sed_environment.htm

11.3 uses of sed

Sed is a versatile tool that can handle a wide range of text processing tasks efficiently and is an essential part of any text-processing toolkit. sed (stream editor) is a powerful Unix tool used for parsing and transforming text. Its applications are diverse and can handle a variety of text-processing tasks. Here are some common and practical uses of sed:

- ✓ **Substitution**: Replace text patterns.
- ✓ **In-Place Editing**: Modify files directly.
- ✓ **Deleting Lines**: Remove lines matching patterns.
- ✓ **Printing**: Output specific lines.
- ✓ **Inserting**: Add text before or after specific lines.
- ✓ **Changing Text**: Modify entire lines.
- ✓ **Regular Expressions**: Handle complex text patterns.
- ✓ **Multiple Commands**: Apply several transformations in one go.
- ✓ **Transforming Text**: Change text case and format.
- ✓ **Handling Delimiters**: Adjust delimiters in data files.
- ✓ **Text Extraction**: Extract parts of files based on patterns.
- ✓ **Combining with Other Commands**: Integrate sed with other tools in pipelines.

11.4 Sed operation

By understanding and utilizing these sed operations, you can efficiently handle various text-processing tasks directly from the command line.

sed (Stream Editor) is a command-line tool used for parsing and transforming text. It operates on streams of text or files, allowing you to perform various operations such as searching, replacing, deleting, and inserting text. Here’s an overview of common sed operations and how they work:

- ✓ **Substitution**: s/pattern/replacement/flags
- ✓ **In-Place Editing**: -i
- ✓ **Deleting Lines**: /pattern/d

- ✓ **Printing Lines**: -n '/pattern/p'
- ✓ **Inserting/Appending Text**: i (before), a (after)
- ✓ **Changing Text**: s/pattern/replacement/
- ✓ **Regular Expressions**: Use regex for complex patterns
- ✓ **Multiple Commands**: -e or semicolons
- ✓ **Text Transformation**: \U for uppercase, \L for lowercase
- ✓ **Handling Delimiters**: Adjust delimiters and formatting
- ✓ **Text Extraction**: Extract lines based on patterns

1. Substitution

Replace text matching a pattern with a replacement string.

Syntax:

sed 's/pattern/replacement/flags' file

Examples:

- Replace the first occurrence of "foo" with "bar":
  $ sed 's/foo/bar/' file.txt
- Replace all occurrences of "foo" with "bar":
  $ sed 's/foo/bar/g' file.txt
- Replace the first occurrence on lines 2 to 4:
  $ sed '2,4s/foo/bar/' file.txt

2. In-Place Editing

Edit files directly and optionally create a backup.

Syntax:

sed -i[extension] 's/pattern/replacement/' file

Examples:

- Edit file.txt in place:
  $ sed -i 's/foo/bar/' file.txt
- Edit with a backup of the original file:
  bash
  Copy code
  $ sed -i.bak 's/foo/bar/' file.txt

3. Deleting Lines

Remove lines that match a specific pattern.

Syntax:

sed '/pattern/d' file

Examples:

- Delete lines containing "debug":
  $ sed '/debug/d' file.txt
- Delete lines from 2 to 4:
  $ sed '2,4d' file.txt

4. Printing Lines

Control which lines are printed from the input.

Syntax:

sed -n '/pattern/p' file

Examples:

- Print lines containing "error":
  $ sed -n '/error/p' file.txt
- Print lines 2 to 4:
  $ sed -n '2,4p' file.txt

5. Inserting and Appending Text

Add text before or after lines matching a pattern.

Insert Text Before: Syntax:

sed '/pattern/i text_to_add' file

Example:

- Insert "Header" before lines containing "Start":
  $ sed '/Start/i Header' file.txt

Append Text After: Syntax:

sed '/pattern/a text_to_add' file

Example:

- Add "Footer" after lines containing "End":
  $ sed '/End/a Footer' file.txt

6. Changing Text

Replace entire lines based on a pattern.

Syntax:

sed 's/pattern/replacement/' file

Examples:

- Change lines containing "OldText" to "NewText":
  $ sed 's/OldText/NewText/' file.txt

7. Using Regular Expressions

Apply regular expressions for complex pattern matching.

Syntax:

sed 's/regex/replacement/flags' file

Examples:

- Replace any sequence of digits with "NUMBER":

  $ sed 's/[0-9]\+/NUMBER/g' file.txt

8. Combining Multiple Commands

Apply several sed commands in one go.

Using -e: Syntax:

sed -e 'command1' -e 'command2' file

Example:

- Replace "foo" with "bar" and then delete lines containing "baz":

  $ sed -e 's/foo/bar/g' -e '/baz/d' file.txt

Using Semicolons: Syntax:

sed 'command1; command2' file

Example:

- Replace "foo" with "bar" and delete lines containing "baz":

  $ sed 's/foo/bar/g; /baz/d' file.txt

9. Transforming Text

Convert text to uppercase or lowercase.

Convert to Uppercase: Syntax:

sed 's/.*/\U&/'

Example:

- Convert entire lines to uppercase:

  $ sed 's/.*/\U&/' file.txt

Convert to Lowercase: Syntax:

sed 's/.*/\L&/'

Example:

- Convert entire lines to lowercase:

  $ sed 's/.*/\L&/' file.txt

10. Handling Delimiters

Change delimiters or adjust formatting.

Syntax:

sed 's/old_delimiter/new_delimiter/g' file

Example:

- Replace commas with tabs in a CSV file:

  $ sed 's/,/\t/g' file.csv

11. Extracting Text

Extract specific parts of text.

Syntax:

sed -n '/start_pattern/,/end_pattern/p' file

Example:

- Extract lines between "Start" and "End":

  $ sed -n '/Start/,/End/p' file.txt

11.5 Pattern addressing

Pattern addressing in sed allows you to perform precise text manipulations by targeting specific lines or ranges of lines based on their content or position. This makes sed a powerful tool for complex text processing tasks.In sed, pattern addressing is a way to specify which lines in a file or input stream should be affected by a command. This allows you to target specific lines or ranges of lines for various operations. Here's a overview of pattern addressing in sed:

- **Single Line Addressing:** N (where N is the line number) or /pattern/
- **Range of Lines:** N1,N2 (line number range) or /pattern1/,/pattern2/
- **Multiple Patterns:** /pattern1\|pattern2/
- **Regular Expressions:** sed '/regex/command/'
- **With -n Flag:** Control line output explicitly

1. Single Lines Addressing

You can address individual lines using line numbers or patterns.

By Line Number:

- **Example:** Apply a command to line 3:

  $ sed '3s/foo/bar/' file.txt

  This command replaces "foo" with "bar" only on line 3.

By Pattern:

- **Example:** Apply a command to lines matching a pattern:

  $ sed '/pattern/s/foo/bar/' file.txt

  This command replaces "foo" with "bar" on lines that contain the pattern "pattern".

2. Addressing Range of Lines

You can specify a range of lines to apply a command using line numbers or patterns.

By Line Number Range:

- **Example:** Apply a command from line 2 to line 5:

  $ sed '2,5s/foo/bar/' file.txt

  This replaces "foo" with "bar" in lines 2 through 5.

By Pattern Range:

- **Example:** Apply a command from the line containing "start" to the line containing "end":

  $ sed '/start/,/end/s/foo/bar/' file.txt

  This replaces "foo" with "bar" from the line containing "start" to the line containing "end", inclusive.

3. Addressing Multiple Patterns

You can combine multiple patterns to address lines.

Using Multiple Patterns:

- **Example:** Apply a command to lines containing either "pattern1" or "pattern2":

  $ sed '/pattern1\|pattern2/s/foo/bar/' file.txt

  This replaces "foo" with "bar" on lines containing either "pattern1" or "pattern2".

4. Addressing Using Regular Expressions

You can use regular expressions to match complex patterns.

Example: Apply a command to lines matching a regular expression:

$ sed '/^start.*end$/s/foo/bar/' file.txt

This replaces "foo" with "bar" on lines that start with "start" and end with "end".

5. Addressing with the -n Flag

When using the -n flag, sed suppresses automatic printing of pattern space. You can then specify exactly which lines to print using addressing.

Example: Print only lines that contain "pattern":

$ sed -n '/pattern/p' file.txt

This prints only the lines that contain "pattern".

6. Addressing with Multiple Commands

You can combine multiple commands and address ranges.

Example: Replace "foo" with "bar" and delete lines containing "baz" from line 2 to line 5:

$ sed '2,5s/foo/bar/; /baz/d' file.txt

7. Examples of Pattern Addressing

1. Replace "foo" with "bar" only on the first line:

$ sed '1s/foo/bar/' file.txt

2. Delete lines from line 10 to the end of the file:

$ sed '10,$d' file.txt

3. Insert "Header" before the first line containing "start":

$ sed '/start/i Header' file.txt

4. Append "Footer" after the last line containing "end":

$ sed '/end/a Footer' file.txt

11.6 Regular expressions

Regular expressions (regex) are powerful tools for pattern matching and text manipulation in many programming languages and text-processing tools, including sed. Regular expressions in sed offer a wide range of text processing capabilities, including:

- ✓ Literal matching
- ✓ Metacharacters for more complex patterns
- ✓ Character classes and predefined classes for grouping and alternatives
- ✓ Quantifiers for specifying the number of matches
- ✓ Anchors for defining positions in the text

Basic Components of Regular Expressions

1. **Literal Characters:**

Match exact characters

sed 's/foo/bar/' file.txt

Matches "foo" and replaces it with "bar".

2. **Metacharacters:**

Matches any single character except a newline.

sed 's/f..r/bar/' file.txt

Matches "f" followed by any two characters and then "r".

Matches the beginning of a line.

sed '/^start/p' file.txt

Matches lines that start with "start".

$: Matches the end of a line.

sed '/end$/p' file.txt

Matches lines that end with "end".

*: Matches zero or more of the preceding element.

sed 's/fo*bar/baz/' file.txt

Matches "f" followed by zero or more "o"s and then "bar".

+: Matches one or more of the preceding element (requires extended regex).

sed -E 's/fo+bar/baz/' file.txt

Matches "f" followed by one or more "o"s and then "bar".

?: Matches zero or one of the preceding element (also requires extended regex).

sed -E 's/fo?bar/baz/' file.txt

Matches "f" followed by zero or one "o" and then "bar".

3. **Character Classes:**

[abc]: Matches any single character within the brackets.

sed 's/[aeiou]/x/' file.txt

Replaces any vowel with "x".

[^abc]: Matches any single character not in the brackets.

sed 's/[^0-9]/x/' file.txt

Replaces any non-digit character with "x".

[a-z]: Matches any single character in the specified range.

sed 's/[a-z]/x/' file.txt

Replaces any lowercase letter with "x".

[0-9]: Matches any digit.

sed 's/[0-9]/x/' file.txt

Replaces any digit with "x".

4. **Predefined Character Classes:**

\d: Matches any digit (requires extended regex).

sed -E 's/\d/x/' file.txt

Replaces any digit with "x".

\D: Matches any non-digit (requires extended regex).

sed -E 's/\D/x/' file.txt

Replaces any non-digit character with "x".

\w: Matches any word character (alphanumeric and underscore) (requires extended regex).

sed -E 's/\w/x/' file.txt

Replaces any word character with "x".

\W: Matches any non-word character (requires extended regex).

sh

Copy code

sed -E 's/\W/x/' file.txt

Replaces any non-word character with "x".

\s: Matches any whitespace character (requires extended regex).

sed -E 's/\s/x/' file.txt

Replaces any whitespace character with "x".

\S: Matches any non-whitespace character (requires extended regex).

sed -E 's/\S/x/' file.txt

Replaces any non-whitespace character with "x".

5. **Quantifiers:**

{n}: Matches exactly n occurrences of the preceding element.

sed -E 's/a{3}/x/' file.txt

Matches exactly three consecutive "a"s and replaces them with "x".

{n,}: Matches n or more occurrences.

sed -E 's/a{2,}/x/' file.txt

Matches two or more consecutive "a"s and replaces them with "x".

{n,m}: Matches between n and m occurrences.

sed -E 's/a{2,4}/x/' file.txt

Matches between two and four consecutive "a"s and replaces them with "x".

6. **Groups and Alternation:**

(abc): Groups together elements (requires extended regex).

sed -E 's/(foo|bar)/baz/' file.txt

Matches either "foo" or "bar" and replaces them with "baz".

|: Alternation (OR) for matching one of several patterns (requires extended regex).

sed -E 's/(foo|bar)/baz/' file.txt

Replaces "foo" or "bar" with "baz".

7. **Anchors and Boundaries:**

\b: Matches a word boundary (requires extended regex).

sed -E 's/\bword\b/x/' file.txt

Matches the word "word" as a whole word and replaces it with "x".

- **\B**: Matches a non-word boundary (requires extended regex).

  sed -E 's/\Bword\B/x/' file.txt

  Matches "word" only if it is not at a word boundary.

Example Commands Using Regular Expressions

1. **Replacing Text Matching a Pattern**

   sed 's/foo/bar/g' file.txt

   Replaces all occurrences of "foo" with "bar" in file.txt.

2. **Deleting Lines Matching a Pattern**

   sed '/pattern/d' file.txt

   Deletes all lines containing "pattern".

3. **Inserting Text Before a Pattern**

   sed '/pattern/i\Inserted text' file.txt

   Inserts "Inserted text" before lines containing "pattern".

4. **Appending Text After a Pattern**

   sed '/pattern/a\Appended text' file.txt

   Appends "Appended text" after lines containing "pattern".

5. **Printing Lines Containing a Pattern**

   sed -n '/pattern/p' file.txt

   Prints only lines that contain "pattern".

11.7 Line information

In sed, line information and manipulation involve various commands and techniques for handling and processing text on a per-line basis. Here's an overview of how to work with line information in sed:

Basic Line Commands

1. **Printing Specific Lines**
   - sed can print specific lines using address patterns. For example:

     sed -n '3p' file.txt

     This command prints only the third line of file.txt.
2. **Range of Lines**
   - You can specify a range of lines to operate on:

     sed -n '2,4p' file.txt

     This prints lines 2 through 4 of file.txt.
3. **All Lines**
   - To print all lines, you can use:

     sed -n 'p' file.txt

     Or simply run sed without -n if you want to apply other commands.

Addressing and Manipulating Lines

1. **Addressing Lines**
   - Addresses in sed specify which lines a command should apply to. You can use:
     - **Line Numbers**: 1, 2, 5, etc.
     - **Patterns**: /pattern/
     - **Range of Line Numbers**: 2,5
     - **Relative Addresses**: .+ (next line), .- (previous line)

   sed '/pattern/p' file.txt

This prints lines that match pattern.

2. **Substituting Text**
    - Substitute text in specific lines or globally:

      sed '3s/old/new/' file.txt

      This substitutes old with new only on line 3.

sed 's/old/new/g' file.txt

This globally substitutes old with new on all lines.

3. **Deleting Lines**
    - Delete specific lines or lines matching a pattern:

      sh

      sed '3d' file.txt

      This deletes the third line.

sed '/pattern/d' file.txt

This deletes all lines containing pattern.

4. **Inserting and Appending Text**
    - Insert text before a specific line:

      sed '/pattern/i\New text' file.txt

      Inserts "New text" before lines matching pattern.
    - Append text after a specific line:

      sh

      Copy code

      sed '/pattern/a\New text' file.txt

      Appends "New text" after lines matching pattern.

Working with Line Numbers

1. **Printing Line Numbers**
    - To include line numbers in the output, use:

      sed = file.txt | sed 'N;s/\n/ /'

      This command adds line numbers to each line.

2. **Using Line Numbers in Commands**
    - Line numbers can be used to perform operations on specific lines:

      sed '2,5d' file.txt

      Deletes lines 2 through 5.

Hold Space and Pattern Space Operations

1. **Copying Between Spaces**
   - Copy pattern space to hold space with h:

     sed 'h;3d;x' file.txt

     This copies the pattern space to the hold space, deletes the third line, and

     exchanges the hold space with the pattern space.
2. **Appending to Hold Space**
   - Append pattern space to hold space with H:

     sed 'H;2,4d;x' file.txt

     This appends the pattern space to the hold space, deletes lines 2 through 4, and exchanges the hold space with the pattern space.
3. **Using G to Append Hold Space**
   - Append hold space to pattern space with G:

     sed 'h;G' file.txt

     Copies the pattern space to hold space and then appends the hold space to the pattern space.

Example

1. **Print Line Numbers with Content**

   sed = file.txt | sed 'N;s/\n/ /'
2. **Delete Lines Matching a Pattern**

   sed '/^#/d' file.txt
3. **Insert Text Before a Line Matching a Pattern**

   sed '/pattern/i\Inserted text' file.txt
4. **Append Text After a Line Matching a Pattern**

   sed '/pattern/a\Appended text' file.txt

11.8 I/O processing

In the sed (stream editor) tool, input and output processing involves several key steps and concepts:

Input Processing

1. **Reading Input:**
   - sed reads input line by line from files or standard input. You can specify the input file directly, or if no file is given, sed reads from standard input (stdin).

   sed 's/old/new/' inputfile

   This command reads inputfile, replaces occurrences of old with new, and outputs the result.

2. **Pattern Space:**
    - Each line of input is loaded into the **pattern space**. This is where sed performs its text manipulations based on the commands provided.
3. **Hold Space:**
    - sed also has a **hold space** used for temporary storage of text. Commands can move data between the pattern space and hold space.

Output Processing

1. **Command Execution:**
    - sed processes each line based on the commands specified. Commands can modify the pattern space, delete lines, or perform various substitutions and transformations.

    sed 's/foo/bar/g'

    This command globally replaces foo with bar in each line of the input.
2. **Output:**
    - After processing the input, sed sends the resulting lines to standard output (stdout) unless redirected otherwise.

    sed 's/foo/bar/' inputfile > outputfile

    This command processes inputfile and writes the results to outputfile.
3. **In-Place Editing:**
    - sed can also edit files directly with the -i option, applying changes to the file without needing redirection.

    sed -i 's/foo/bar/' inputfile

    This command modifies inputfile in place by replacing foo with bar.

11.9 Yanking

In sed, "yanking" refers to copying text from the pattern space into the hold space. This is akin to the yank command in text editors like vi, which allows you to copy text to a buffer.

Here's a breakdown of how yanking is handled in sed:

Commands Involving Yank

1. **Yanking to the Hold Space:**

    To copy (or "yank") the entire content of the pattern space to the hold space, you use the h command. This command saves the current pattern space into the hold space.

    sed 'h'

    This command copies the content of the pattern space into the hold space, which can then be used for further processing.
2. **Yanking from the Hold Space:**

    To retrieve (or "yank") the content from the hold space back into the pattern space, use the g command. This replaces the current pattern space with the content of the hold space.

    sed 'g'

This command gets the content from the hold space and places it into the pattern space.

3. **Appending Hold Space Content:**

   To append the content of the hold space to the pattern space, use the G command. This concatenates the content of the hold space to the current pattern space, with a newline separating the two.

   sed 'G'

   This appends the content of the hold space to the pattern space, separated by a newline.

4. **Appending Pattern Space Content:**

   To append the content of the pattern space to the hold space, use the H command. This command appends the pattern space to the hold space, with a newline separator.

   sed 'H'

   This appends the pattern space to the hold space, with the new content separated by a newline.

Example Usage

Let's say you have a file called file.txt with the following content:

Copy code

line1

line2

line3

Example 1: Copying Pattern Space to Hold Space

sed -n '1{h;p;}' file.txt

This command copies the first line (line1) to the hold space and prints it. The -n option suppresses automatic printing, and the 1{h;p;} part means:

- 1 — On the first line,
- {h;p;} — Copy (h) the pattern space to the hold space and print (p) the pattern space.

Example 2: Appending Pattern Space to Hold Space

sed 'H;G' file.txt

This command:

- H appends each line of the pattern space to the hold space (with newlines).
- G appends the hold space content (with a newline) to the pattern space.

In this case, the hold space will accumulate all lines of the file, each separated by newlines.

11.10 Putting:

In sed, the concept of "putting" text into a file or a specific location isn't directly named as such, but it can be achieved using certain commands. Here's a detailed explanation of how to effectively "put" or manage text using sed:

Commands for "Putting" Text in sed

1. **w Command: Write to a File**

- The w command is used to write the current pattern space to a file. This is useful for directing output to a specific file.

sed '2w output.txt' file.txt

This command writes the second line of file.txt to output.txt.

2. **a Command: Append Text After a Line**
   - The a (append) command is used to add text after a line that matches a specific pattern.

sed '/pattern/a\new text' file.txt

This appends "new text" after every line that matches the pattern.

3. **i Command: Insert Text Before a Line**
   - The i (insert) command inserts text before a line that matches a specific pattern.

sed '/pattern/i\new text' file.txt

This inserts "new text" before every line that matches the pattern.

4. **r Command: Read Text from a File**
   - The r (read) command reads lines from another file and inserts them after the current line in the pattern space.

sed '/pattern/r otherfile.txt' file.txt

This inserts the content of otherfile.txt after every line in file.txt that matches the pattern.

5. **G Command: Append Hold Space to Pattern Space**
   - The G command appends the content of the hold space to the pattern space, separated by a newline.

sed 'h; G' file.txt

This command first copies the pattern space to the hold space (h) and then appends the hold space content to the pattern space (G).

6. **H Command: Append Pattern Space to Hold Space**
   - The H command appends the content of the pattern space to the hold space, separated by a newline.

sed 'H; x' file.txt

This command appends the pattern space to the hold space and then exchanges the hold space with the pattern space (x).

Example Use Cases

Example 1: Appending Text to Specific Lines

sed '/^line2/a\This is an appended line' file.txt

This command appends "This is an appended line" after every line that starts with "line2".

Example 2: Inserting Text Before Specific Lines

sed '/^line2/i\This is an inserted line' file.txt

This command inserts "This is an inserted line" before every line that starts with "line2".

Example 3: Writing to an Output File

sed '2p' file.txt > output.txt

This command prints the second line of file.txt and redirects it to output.txt.

Example 4: Reading from Another File

sed '/^line2/r additional.txt' file.txt

This command reads and inserts the content of additional.txt after every line that starts with "line2" in file.txt.

11.11 Branching commands

The branching commands b, t, and T enable changing the flow of sed programs.

By default, sed reads an input line into the pattern buffer, then continues to processes all commands in order. Commands without addresses affect all lines. Commands with addresses affect only matching lines.

sed does not support a typical if/then construct. Instead, some commands can be used as conditionals or to change the default flow control:

d

delete (clears) the current pattern space, and restart the program cycle without processing the rest of the commands and without printing the pattern space.

D

delete the contents of the pattern space *up to the first newline,* and restart the program cycle without processing the rest of the commands and without printing the pattern space.

[addr]X

[addr]{ X ; X ; X }

/regexp/X

/regexp/{ X ; X ; X }

Addresses and regular expressions can be used as an if/then conditional: If *[addr]* matches the current pattern space, execute the command(s). For example: The command /^#/d means: *if* the current pattern matches the regular expression ^# (a line starting with a hash), *then* execute the d command: delete the line without printing it, and restart the program cycle immediately.

b

Branch unconditionally (that is: always jump to a label, skipping or repeating other commands, without restarting a new cycle). Combined with an address, the branch can be conditionally executed on matched lines.

t

Branch conditionally (that is: jump to a label) *only if* a s/// command has succeeded since the last input line was read or another conditional branch was taken.

T

similar but opposite to the t command: branch only if there has been *no* successful substitutions since the last input line was read.

The following two sed programs are equivalent. The first (contrived) example uses the b command to skip the s/// command on lines containing ‘1’. The second example uses an address with negation (‘!’) to perform substitution only on desired lines. The y/// command is still executed on all lines:

```
$ printf '%s\n' a1 a2 a3 | sed -E '/1/bx ; s/a/z/ ; :x ; y/123/456/'
a4
z5
z6
$ printf '%s\n' a1 a2 a3 | sed -E '/1/!s/a/z/ ; y/123/456/'
a4
z5
z6
```

11.12 Multiline input processing

The commands D, G, H, N, and P can handle multiple lines as a single buffer. They function similarly to their lowercase counterparts (d, g, h, n, p), but with the added capability to append or remove lines while accounting for embedded newlines. This means you can add or remove lines from both the pattern space and the hold space while preserving the structure of the data.

They operate as follows:

D

deletes line from the pattern space until the first newline, and restarts the cycle.

G

appends line from the hold space to the pattern space, with a newline before it.

H

appends line from the pattern space to the hold space, with a newline before it.

N

appends line from the input file to the pattern space.

P

prints line from the pattern space until the first newline.

The following example illustrates the operation of N and D commands:

```
$ seq 6 | sed -n 'N;l;D'
1\n2$
2\n3$
3\n4$
4\n5$
5\n6$
```

1. sed starts by reading the first line into the pattern space (i.e. ‘1’).

2. At the beginning of every cycle, the N command appends a newline and the next line to the pattern space (i.e. ‘1’, ‘\n’, ‘2’ in the first cycle).
3. The l command prints the content of the pattern space unambiguously.
4. The D command then removes the content of pattern space up to the first newline (leaving ‘2’ at the end of the first cycle).
5. At the next cycle the N command appends a newline and the next input line to the pattern space (e.g. ‘2’, ‘\n’, ‘3’).

A common technique to process blocks of text such as paragraphs (instead of line-by-line) is using the following construct:

sed '/./{H;$!d} ; x ; s/REGEXP/REPLACEMENT/'

1. The first expression, /./{H;$!d} operates on all non-empty lines, and adds the current line (in the pattern space) to the hold space. On all lines except the last, the pattern space is deleted and the cycle is restarted.
2. The other expressions x and s are executed only on empty lines (i.e. paragraph separators). The x command fetches the accumulated lines from the hold space back to the pattern space. The s/// command then operates on all the text in the paragraph (including the embedded newlines).

The following example demonstrates this technique:

$ cat input.txt

a a a aa aaa

aaaa aaaa aa

aaaa aaa aaa

bbbb bbb bbb

bb bb bbb bb

bbbbbbbb bbb

ccc ccc cccc

cccc ccccc c

cc cc cc cc

$ sed '/./{H;$!d} ; x ; s/^/\nSTART-->/ ; s/$/\n<--END/' input.txt

START-->

a a a aa aaa

aaaa aaaa aa

aaaa aaa aaa

<--END

START-->

bbbb bbb bbb

bb bb bbb bb

bbbbbbbb bbb

<--END

START-->

ccc ccc cccc

cccc ccccc c

cc cc cc cc

<--END

11.13 SUMMARY:

In this unit, we have learn about the basic concept of sed editor , uses of sed, sed operation, standard operations, pattern addressing, regular expressions, line information, I/O processing, yanking, putting, branching commands, multiline input processing

11.14 Check your progress

- **What do you mean sed?**

……
……
……
……

- ✓ **What are the uses of sed ?**

……
……
……
……

- ✓ **Discuss the about regular expression**

……
……
……
……
…….

- ✓ **Discuss about branching in sed**

……
……
……
……

11.15 Further Reading

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/
- ✓ https://www.tutorialspoint.com/sed/sed_environment.htm
- ✓ https://www.gnu.org/software/sed/manual/sed.html
- ✓ https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/

UNIT-12 : Bash scripting-I

Structure

12.1. Introduction to Bash scripting

Bash is a Unix shell and command language commonly used across various operating systems, and it serves as the default command interpreter for most Linux distributions. Its name, "Bourne-Again SHell," reflects its evolution from the original Bourne Shell.

You can use Bash interactively in your terminal or employ it for scripting purposes, much like other programming languages. This guide will introduce you to fundamental Bash scripting concepts, including variables, user input, comments, arguments, arrays, conditional expressions, control structures, loops, functions, debugging, and testing.

Bash scripts are particularly useful for automating repetitive tasks, which can significantly save time. For instance, consider a scenario where you and four other developers need to repeatedly set up a development environment. Instead of manually configuring the environment each time, which involves repeating the same process five times, you can streamline the setup with a Bash script. By creating a script with all the necessary commands, you can share this single file with your team. They only need to run the script, and it will automatically handle the environment setup for them.

To start writing Bash scripts, you'll need a UNIX terminal and a text editor such as Sublime Text, VS Code, or a terminal-based editor like vim or nano.

12.1a. Objectives

After studying this unit ,the learner would be able to understand the following:-

- Basic of Bash shell scripting

- Bash scriptingVariables
- variable assignment
- variable scope
- Operators
- Command Line Arguments,
- Setting Values of Positional Parameters,
- Using Shift on Positional Parameters

12.2 Bash Structure

To start, you'll want to create a new file with a .sh extension. For example, let's create a file named uprtou.sh.

You can create this file using the touch command:

touch uprtou.sh

Alternatively, you can use a text editor to create and open the file:

nano uprtou.sh

When writing a Bash script, the first line should specify the absolute path to the Bash interpreter. This is known as the Shebang and looks like this:

#!/bin/bash

The Shebang tells the operating system to use the /bin/bash executable to run the script.

However, since Bash might not always be located in `/bin/bashes—particularly on non-Linux systems or if installed as an optional package—it's often more reliable to use:

#!/usr/bin/env bash

This approach uses env to find the bash executable based on the directories listed in the PATH environment variable, making it more adaptable across different systems.

12.3 First bash script

Once you've created your uprtou.sh file and added the Bash Shebang at the top, you're ready to write your first script.

1. **Edit the File**: Open the uprtou.sh file and add the following lines after the Shebang:

   #!/bin/bash

   echo "Hello World!"

   Save the file and exit the editor.

2. **Make the Script Executable**: Run the following command to give execute permissions to your script:

   chmod +x uprtou.sh

3. **Execute the Script**: Run the script with:

   ./uprtou.sh

You should see the message "Hello World!" displayed on the screen.

Alternatively, you can run the script using the bash command:

bash uprtou.sh

You can also directly execute a similar command in your terminal to achieve the same result:

echo "Hello uprtou!"

Creating scripts is particularly useful for automating tasks by combining multiple commands into one file.

12.4 Variable

In programming, a variable represents a value that can change based on conditions or input. In Bash scripting, a variable can hold numbers, characters, or strings of characters. This article will cover how variables work in Bash scripting.

Here are the key rules for defining variables in Bash scripts:

- **Variable Names**: They can include uppercase and lowercase letters, numbers, and underscores.
- **Naming Conventions**: It's considered good practice to use uppercase letters for variable names in Bash scripts.
- **No Spaces**: Spaces are not allowed in variable names.
- **Reserved Keywords**: You cannot use reserved keywords, such as if or else, as variable names.

The simplest example of the use of variables in Bash scripting can be given as –

Example Script:

```
NAME="chandra"
echo "His name is $NAME"
```

Output:

```
His name is Chandra
```

The above example shows a variable '**name**'. Then it is used with an **echo** command to display its value. So, the basic syntax for writing variables within a Bash Script will be –

Syntax of Writing Variables:

```
VariableName=value
echo $VariableName #for accessing the value
```

If we want to change the value of a variable then we can do that. The basic syntax for that will be –

```
VariableName=value
VariableName=newValue
echo $VariableName #for accessing the value
```

Below is an example of the same –

Example Script:

```
AGE=10                  #assigning      a       value
AGE=20                  #changing       the     value
echo        $AGE        #display        the     value
```

Output:

20

12.5 Scope of Variable

In programming, the **scope** of a variable defines where the variable is accessible within the code. In general:

- **Local Variables**: Declared inside a function and are only accessible within that function.
- **Global Variables**: Declared outside of any function and are accessible throughout the entire program or script.

In Bash scripting, the concept of variable scope is slightly different:

- **Global Variables**: By default, any variable declared in a Bash script, whether inside or outside of a function, is global and accessible throughout the script.
- **Local Variables**: To create a variable that is only accessible within a specific function, you must use the local keyword.

Here's an example to illustrate both global and local variables in Bash:

```
#!/bin/bash

# Global variable
global_var="I am global"

# Function that uses a local variable
my_function() {
   local local_var="I am local"
   echo "Inside function: $local_var"
}

# Call the function
my_function
```

```
# Try to access the local variable outside the function
echo "Outside function: $local_var"
```

Explanation:

1. **Global Variable**: global_var is declared outside any function and is accessible everywhere in the script.
2. **Local Variable**: local_var is declared inside my_function using the local keyword and is only accessible within that function.

Output:

```
Inside function: I am local
Outside function:
```

Notice that local_var is not accessible outside the function, whereas global_var can be used anywhere in the script.

Another example of the same is given below –

Example Script:

```
#!/bin/bash

var1="Apple" # global variable
myfun(){
   local var2="Banana" #local variable
   var3="Cherry" #global variable
   echo "The name of first fruit is $var1"
   echo "The name of second fruit is $var2"
}
myfun

echo "The name of first fruit is $var1"
# trying to access local variable
echo "The name of second fruit is $var2"
echo "The name of third fruit is $var3"
```

Output of Variable Scope:

```
The name of first fruit is Apple
The name of second fruit is Banana
The name of first fruit is Apple
The name of second fruit is
The name of third fruit is Cherry
```

Here in this above example, var2 is a local variable, so when we are accessing it from the function it is doing fine but when we are trying to access it outside the function, it is giving us an empty result in the output.

On the other hand, unlike programming languages, even though var3 is defined inside a function still it is acting as a global variable and it can be accessed outside the function.

12.6 Variable assignment

In Bash scripts, variable assignment is indeed straightforward but comes with various options to enhance script functionality. Variables in Bash are used to store and manipulate data, and while variable assignment is simple, understanding its scope and usage can provide greater flexibility in your scripts.

Types of Variable Assignments in Bash

1. Global Variables:

- **Definition**: By default, any variable declared outside of a function is a global variable. This means it can be accessed and modified from anywhere within the script.
- **Syntax**:

  variable_name="value"

  Example:

  ```
  # Global variable
  greeting="Hello, World!"


  echo $greeting  # Output: Hello, World!
  ```

2. Local Variables:

- **Definition**: Variables declared inside a function with the local keyword are local to that function. They are not accessible outside the function in which they are declared.
- **Syntax**:

  ```
  my_function() {
     local variable_name="value"
     # Do something with the local variable
  }
  ```

  Example:

  ```
  # Global variable
  global_var="I am global"


  my_function() {
     local local_var="I am local"
     echo "Inside function: $local_var"
  ```

```
}

my_function
echo "Outside function: $local_var"  # This will be empty as local_var is not accessible here
```

Additional Features in Variable Assignment

- **Variable Expansion**: Use `$` to access the value of a variable.

  ```
  name="Alice"
  echo "Hello, $name!"  # Output: Hello, Alice!
  ```

- **Default Values**: You can provide default values using the `${variable:-default}` syntax.

  ```
  echo ${username:-"Guest"}  # Outputs "Guest" if $username is not set
  ```

- **Read-Only Variables**: Use `readonly` to make a variable immutable.

  ```
  readonly pi=3.14
  ```

- **Exporting Variables**: Use `export` to make a variable available to subprocesses.

  ```
  export PATH=$PATH:/new/directory
  ```

Understanding and utilizing these variable assignment features will help you create more robust and flexible Bash scripts.

12.7 Operations on Variables

We can perform both numerical and string operations on variables on Bash scripting. An example of the same is given below –

Example Script:

```
NUM1=10 #variable 1
NUM2=5 #variable 2

# Numerical Operations over the variables
SUM=$(( $NUM1 + $NUM2 ))
SUBTRACT=$(( $NUM1 - $NUM2 ))
MULTIPLY=$(( $NUM1 * $NUM2 ))
DIVIDE=$(( $NUM1 / $NUM2 ))

echo "Addition : $SUM"
echo "Subtraction : $SUBTRACT"
echo "Multiply : $MULTIPLY"
echo "Divide : $DIVIDE"

# String Operations over the variables
```

```
NAME="GeeksforGeeks"
echo ${NAME:0:5} #substring extraction

FIRST_NAME="Isaac"
LAST_NAME="Newton"
echo ${FIRST_NAME}" "${LAST_NAME}
```

Output of Operations on Variables:

Addition : 15

Subtraction : 5

Multiply : 50

Divide : 2

Geeks
Isaac Newton

In the above example, we have performed different numerical operations like addition, subtraction, multiplication, and division using variables and also performed basic string operations like concatenation and substring extraction.

12.8 Special Variables

There are some special types of variables present in the brush. These are also called internal variables or preset variables. Below is the list of some of them.

Internal variables or Preset variables	**Description**
$#	How many command line parameters were passed to the script.
$@	All the command line parameters are passed to the script.
$?	The exit status of the last process to run.
$$	The Process ID (PID) of the current script.

Internal variables or Preset variables	Description
$USER	The username of the user executing the script.
$HOSTNAME	The hostname of the computer running the script.
$SECONDS	The number of seconds the script has been running for.
$RANDOM	Returns a random number.
$LINENO	Returns the current line number of the script.

12.9 Operators

In Bash/shell scripting, operators are used to perform operations on variables and values. Here are the five basic types of operators commonly used in Bash scripts:

Arithmetic Operators:

- ✓ Addition (+): Binary operation used to add two operands.
- ✓ Subtraction (-): Binary operation used to subtract two operands.
- ✓ Multiplication (*): Binary operation used to multiply two operands.
- ✓ Division (/): Binary operation used to divide two operands.
- ✓ Modulus (%): Binary operation used to find remainder of two operands.
- ✓ Increment Operator (++): Unary operator used to increase the value of operand by one.
- ✓ Decrement Operator (–): Unary operator used to decrease the value of a operand by one

These operators perform mathematical operations.

- **Addition**: +

  result=$((5 + 3)) # result will be 8
- **Subtraction**: -

  result=$((5 - 3)) # result will be 2

- **Multiplication**: *

  result=$((5 * 3)) # result will be 15
- **Division**: /

  result=$((15 / 3)) # result will be 5
- **Modulus**: %

  result=$((15 % 4)) # result will be 3

Relational Operators

Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. There are six types:

- **'==' Operator**: Double equal to operator compares the two operands. It returns true if they are equal otherwise returns false.
- **'!=' Operator**: Not equal to operator returns true if the two operands are not equal otherwise it returns false.
- **'<' Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
- **'<=' Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- **'>' Operator:** Greater than operator returns true if the first operand is greater than the second operand otherwise returns false.

'>=' Operator: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

These operators are used for comparing values.

- **Equal to**: -eq

  if ["$a" -eq "$b"]; then

  echo "a is equal to b"

  fi
- **Not equal to**: -ne

  if ["$a" -ne "$b"]; then

  echo "a is not equal to b"

  fi
- **Greater than**: -gt

  if ["$a" -gt "$b"]; then

  echo "a is greater than b"

  fi
- **Less than**: -lt

  if ["$a" -lt "$b"]; then

```
    echo "a is less than b"
fi
```

- **Greater than or equal to**: -ge

```
if [ "$a" -ge "$b" ]; then
    echo "a is greater than or equal to b"
fi
```

- **Less than or equal to**: -le

```
if [ "$a" -le "$b" ]; then
    echo "a is less than or equal to b"
fi
```

3. Logical Operators

These operators are used to perform logical operations.

- **AND**: &&

```
if [ "$a" -eq "$b" ] && [ "$c" -eq "$d" ]; then
    echo "Both conditions are true"
fi
```

- **OR**: ||

```
if [ "$a" -eq "$b" ] || [ "$c" -eq "$d" ]; then
    echo "At least one condition is true"
fi
```

- **NOT**: !

```
if ! [ "$a" -eq "$b" ]; then
    echo "Condition is false"
fi
```

4. String Operators

These operators are used for string comparison and manipulation.

- **Equal to**: =

```
if [ "$str1" = "$str2" ]; then
    echo "Strings are equal"
fi
```

- **Not equal to**: !=

```
if [ "$str1" != "$str2" ]; then
    echo "Strings are not equal"
fi
```

- **Length of string**: -n and -z

  ```
  if [ -n "$str" ]; then
     echo "String is not empty"
  fi


  if [ -z "$str" ]; then
     echo "String is empty"
  fi
  ```

5. File Test Operators

These operators are used to check file attributes and existence.

- **File exists**: -e

  ```
  if [ -e "$file" ]; then
     echo "File exists"
  fi
  ```

- **File is a directory**: -d

  ```
  if [ -d "$dir" ]; then
     echo "It is a directory"
  fi
  ```

- **File is readable**: -r

  bash

  Copy code

  ```
  if [ -r "$file" ]; then
     echo "File is readable"
  fi
  ```

- **File is writable**: -w

  bash

  Copy code

  ```
  if [ -w "$file" ]; then
     echo "File is writable"
  fi
  ```

- **File is executable**: -x

  ```
  if [ -x "$file" ]; then
     echo "File is executable"
  fi
  ```

Understanding these basic operators will help you perform a wide range of operations and comparisons in your Bash scripts.

12.10 Command-line Arguments:

In a Bash script, command line arguments are a way to pass data to the script when it's executed. This allows you to provide input to the script dynamically rather than hardcoding values into the script itself. Using these concepts, you can make your Bash scripts more flexible and interactive by accepting and processing command line arguments.

- ✓ **$1, $2, ...**: Access specific arguments.
- ✓ **$@**: All arguments as separate words.
- ✓ **$***: All arguments as a single word.
- ✓ **$#**: Number of arguments.
- ✓ **$0**: Name of the script.

Basic Concepts

1. **Accessing Command Line Arguments:**
    - **$1, $2, ...**: Represents the first, second, and subsequent arguments.

      # script.sh

      echo "First argument: $1"

      echo "Second argument: $2"
    - **$@**: Represents all arguments as separate words.

      echo "All arguments: $@"
    - **$***: Represents all arguments as a single word. This is often used with quotes to handle spaces.

      echo "All arguments (as a single string): $*"
    - **$#**: Represents the number of arguments.

      echo "Number of arguments: $#"
    - **$0**: Represents the name of the script itself.

      sh

      Copy code

      echo "Script name: $0"

Examples

1. **Simple Script to Echo Arguments:**

   Create a file named example.sh:

   #!/bin/bash

   echo "Script name: $0"

   echo "First argument: $1"

   echo "Second argument: $2"

```
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Make the script executable and run it:

```
chmod +x example.sh
./example.sh arg1 arg2
```

Output:

yaml

Copy code

```
Script name: ./example.sh
First argument: arg1
Second argument: arg2
All arguments: arg1 arg2
Number of arguments: 2
```

2. **Handling No Arguments:**

   You might want to check if no arguments are provided and handle it accordingly:

```
#!/bin/bash
if [ $# -eq 0 ]; then
  echo "No arguments provided."
  exit 1
fi

echo "Arguments provided: $@"
```

3. **Looping Through Arguments:**

   You can loop through the arguments using "$@":

```
#!/bin/bash
for arg in "$@"; do
  echo "Argument: $arg"
done
```

4. **Using Arguments for Conditional Logic:**

   You can use arguments to drive the logic in your script:

```
#!/bin/bash
if [ "$1" == "start" ]; then
  echo "Starting the process..."
elif [ "$1" == "stop" ]; then
```

```
  echo "Stopping the process..."
else
  echo "Usage: $0 {start|stop}"
  exit 1
fi
```

5. **Positional Parameters and Named Arguments:**

   For more complex scripts, you might use named arguments or flags:

```
#!/bin/bash
while getopts "a:b:" opt; do
  case $opt in
    a) param_a="$OPTARG" ;;
    b) param_b="$OPTARG" ;;
    \?) echo "Invalid option -$OPTARG" ;;
  esac
done

echo "Option a: $param_a"
echo "Option b: $param_b"
```

Run it with:

./script.sh -a value1 -b value2

12.11 Positional parameters

In Bash scripting, **positional parameters** are essential for handling input data passed to scripts. These parameters allow scripts to accept and process arguments provided at runtime, making scripts more flexible and dynamic.

A solid understanding of positional parameters and how to use them effectively is crucial for creating dynamic and flexible Bash scripts. They enable you to write scripts that can handle a variety of inputs and adapt to different situations, automating tasks and enhancing the script's utility.These are special variables that represent the arguments passed to a script. They are accessed using $1, $2, $3, and so on, where $1 is the first argument, $2 is the second, etc.

Example Script Using Positional Parameters

```
#!/bin/bash

# Display the first three positional parameters
echo "First argument: $1"
echo "Second argument: $2"
```

```
echo "Third argument: $3"


# Display the total number of arguments
echo "Total number of arguments: $#"
```

How to Run the Script:

Save the script to a file, e.g., example.sh, and make it executable:

```
chmod +x example.sh
```

Run the script with arguments:

```
./example.sh arg1 arg2 arg3
```

Output:

```
yaml
Copy code
First argument: arg1
Second argument: arg2
Third argument: arg3
Total number of arguments: 3
```

Variations and Advanced Uses

1. **Accessing All Arguments**: Use $@ or $* to refer to all positional parameters.
   - **$@**: Treats each argument as a separate word, which is useful when iterating over arguments.

     ```
     for arg in "$@"; do
       echo "$arg"
     done
     ```

   - **$***: Treats all arguments as a single word, which can be useful in certain contexts but is less commonly used.
2. **Default Values**: You can set default values for arguments if they are not provided.

   ```
   arg1=${1:-default_value}
   ```

3. **Handling Variable Number of Arguments**: You can process an unknown number of arguments using loops.

   ```
   while [ "$1" != "" ]; do
     echo "Processing argument: $1"
     shift
   done
   ```

4. **Using getopts for Option Parsing**: For more complex scripts, use getopts to handle options and flags.

   ```
   while getopts "a:b:c:" opt; do
   ```

```
    case $opt in
        a) paramA=$OPTARG ;;
        b) paramB=$OPTARG ;;
        c) paramC=$OPTARG ;;
        *) echo "Invalid option" ;;
    esac
done
```

5. **Accessing Script Name**: Use `$0` to get the name of the script itself.

   ```
   echo "Script name: $0"
   ```

12.12 Shift on positional parameter

The shift command in Bash scripting is used to manipulate the positional parameters. It allows you to process command-line arguments in a script by shifting them to the left, effectively discarding the first argument and making the next one the new $1. This is especially useful when you don't know the number of arguments in advance and need to process each argument sequentially.

How the shift Command Works

- **Basic Usage**: The shift command shifts the positional parameters to the left by one position. For example, after executing shift, $2 becomes $1, $3 becomes $2, and so on.
- **Shift with a Number**: You can specify a number to shift by that many positions. For example, shift 5 will make $6 become $1, $7 become $2, and so forth.

Example of Using shift

Here's a simple example demonstrating how shift can be used to process an unknown number of command-line arguments:

```
#!/bin/bash

# Initial number of arguments
echo "Total arguments: $#"

# Process each argument
while [ "$#" -gt 0 ]; do
    echo "Processing argument: $1"
    shift
done

echo "No more arguments"
```

How to Run the Script:

Save the script to a file, e.g., shift_example.sh, and make it executable:

chmod +x shift_example.sh

Run the script with multiple arguments:

./shift_example.sh arg1 arg2 arg3 arg4

Output:

yaml

Copy code

Total arguments: 4

Processing argument: arg1

Processing argument: arg2

Processing argument: arg3

Processing argument: arg4

No more arguments

Key Points

- **Shifting by a Number**: To shift by a specific number of positions, use shift N, where N is the number of positions to shift. For example:

  shift 2

  This will move $3 to $1, $4 to $2, etc.

- **Looping Through Arguments**: Typically, shift is used within a loop to process all arguments. The while loop condition ["$#" -gt 0] checks if there are any arguments left to process.
- **Arguments Reduction**: Each time shift is executed, the total number of arguments ($#) is reduced by the number of positions shifted, making it easier to handle variable-length argument lists.

By using the shift command effectively, you can create scripts that handle a flexible number of arguments and perform actions based on them dynamically

12.13 SUMMARY

In this unit, we have learn about the basic concept of Bash scripting:Variables- variable assignment and variable scope, Operators, Command Line Arguments, Setting Values of Positional Parameters, Using Shift on Positional Parameters

12.14 Check your progress:

- **What do you mean variable assignment in Bash script?**

……
……
……
……

- ✓ **What are the scope of variable?**

……
……
……
……

✓ **Discuss the about Positional Parameters**

……
……
……
……

✓ **Discuss about Command Line Arguments**

……
……
……
……

12.15 Further Reading

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/
- ✓ https://www.tutorialspoint.com
- ✓ https://www.gnu.org

Unit -13 : Bash scripting-II

Structure

13.1 Introduction

Control flow statements are crucial in programming as they allow you to manage the execution path of your code based on conditions and loops. They provide flexibility in controlling the sequence of statements and making decisions during runtime. Control flow statements are essential for creating flexible and dynamic programs. They allow you to:

- **Make Decisions**: Using if, else, and switch statements to execute code based on conditions.
- **Repeat Actions**: Using for, while, and do-while loops to perform actions multiple times.
- **Modify Execution**: Using break, continue, and return to control the flow of execution within loops and functions.
- **Handle Errors**: Using exception handling mechanisms to manage runtime errors and exceptional conditions.

Understanding and effectively using these control flow constructs is key to writing efficient and robust code.

13.1a. Objectives

After studying this unit ,the learner would be able to understand the following:-

- Basic concept of Control Flow in scripting
- Types of Control Flow statements
- Decision Statements used in scripting
- How to use loops and case statements in shell scripting
- Arithmetic used in Shell Script,
- Concept of Array in shell script
- File and String Tests

13.2 Types of Control Flow statements

Control Flow Statements Type	Control Flow Statement	Description
Conditional Statements	if-else	Executes a block of code if a specified condition is true, and another block if the condition is false.
	switch-case	Evaluates a variable or expression and executes code based on matching cases.
Looping Statements	for	Executes a block of code a specified number of times, typically iterating over a range of values.
	while	Executes a block of code as long as a specified condition is true.
	do-while	Executes a block of code once and then repeats the execution as long as a specified condition is true.
Jump Statements	break	Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.
	continue	Skips the current iteration of a loop and continues with the next iteration.

Control Flow Statements Type	Control Flow Statement	Description
	return	Exits a function and returns a value to the caller.
	goto	Transfers control to a labeled statement within the same function. (Note: goto is generally discouraged due to its potential for creating unreadable and error-prone code.)

13.3 If statement

The format of an if statement in Bash is as follows:

```
if [[ some_test ]]
then
  <commands>
 fi
```

Example :

```
#!/bin/bash
# Bash if statement example
read -p "What is your name? " name
if [[ -z ${name} ]]
 then
echo "Please enter your name!"
fi
```

13.4 If Else statement

With an if-else statement, you can specify an action in case that the condition in the if statement does not match. We can combine this with the conditional expressions from the previous section as follows:

```
#!/bin/bash
# Bash if statement example
read -p "What is your name? " name
if [[ -z ${name} ]]
then    echo "Please enter your name!"
else    echo "Hi there ${name}"
fi
```

You can use the above if statement with all of the conditional expressions .

```
#!/bin/bash
admin="devdojo"
read -p "Enter your username? " username
# Check if the username provided is the admin
if [[ "${username}" == "${admin}" ]] ; then
 echo "You are the admin user!"
 else
 echo "You are NOT the admin user!"
 fi
```

Here is another example of an if statement which would check your current User ID and would not allow you to run the script as the root user:

```
#!/bin/bash
if (( $EUID == 0 )); then
 echo "Please do not run as root"
  exit
  fi
```

If you put this on top of your script it would exit in case that the EUID is 0 and would not execute the rest of the script.

You can also test multiple conditions with an if statement. In this example we want to make sure that the user is neither the admin user or the root user to ensure the script is incapable of causing too much damage. We'll use the or operator in this example, noted by ||. This means that either of the conditions needs to be true. If we used the and operator of && then both conditions would need to be true.

```
#!/bin/bash
admin="devdojo"
read -p "Enter your username? " username
# Check if the username provided is the admin
if [[ "${username}" != "${admin}" ]] || [[ $EUID != 0 ]] ;
then
 echo "You are not the admin or root user, but please be safe!"
else
echo "You are the admin user! This could be very destructive!"
fi
```

If you have multiple conditions and scenerios, then can use elif statement with if and else statements.

```
#!/bin/bash
read -p "Enter a number: " num
if [[ $num -gt 0 ]] ; then
```

```
 echo "The number is positive"
elif [[ $num -lt 0 ]] ; then
echo "The number is negative"
else
echo "The number is 0"
fi
```

13.5 Switch case statements

As in other programming languages, you can use a case statement to simplify complex conditionals when there are multiple different choices. So rather than using a few if, and if-else statements, you could use a single case statement.

The Bash case statement syntax looks like this:

```
case $some_variable in
 pattern_1)    commands
  ;;
 pattern_2| pattern_3)
   commands
    ;;
 *)
  default commands
 ;;
 esac
```

A quick rundown of the structure:

All case statements start with the case keyword.

On the same line as the case keyword, you need to specify a variable or an expression followed by the in keyword.

After that, you have your case patterns, where you need to use) to identify the end of the pattern.

You can specify multiple patterns divided by a pipe: |.

After the pattern, you specify the commands that you would like to be executed in case that the pattern matches the variable or the expression that you've specified.

All clauses have to be terminated by adding ;; at the end.

You can have a default statement by adding a * as the pattern.

To close the case statement, use the esac (case typed backwards) keyword.

Example of a Bash case statement:

```
#!/bin/bash
read -p "Enter the name of your car brand: " car
```

```
case $car  in
 Tesla)
  echo -n "${car}'s car factory is in the USA."
  ;;
 BMW | Mercedes | Audi | Porsche)
  echo -n "${car}'s car factory is in Germany."
  ;;
 Toyota | Mazda | Mitsubishi | Subaru)
  echo -n "${car}'s car factory is in Japan."
  ;;
 *)
 echo -n "${car} is an unknown car brand"
   ;;
esac
```

With this script, we are asking the user to input a name of a car brand like Telsa, BMW, Mercedes and etc.

Then with a **case** statement, we check the brand name and if it matches any of our patterns, and if so, we print out the factory's location.

If the brand name does not match any of our case statements, we print out a default message: an unknown car brand.

13.6 Loops

As with any other language, loops are very convenient. With Bash you can use for loops, while loops, and until loops

13.6.1 For loops

Here is the structure of a for loop:

```
for var in ${list}
do
 your_commands
  done
```

Example:

```
#!/bin/bash
users="devdojo bobby tony"
for user in ${users}
 do
echo "${user}"
```

done

example:

First, we specify a list of users and store the value in a variable called $users.

After that, we start our for loop with the for keyword.

Then we define a new variable which would represent each item from the list that we give. In our case, we define a variable called user, which would represent each user from the $users variable.

Then we specify the in keyword followed by our list that we will loop through.

On the next line, we use the do keyword, which indicates what we will do for each iteration of the loop.

Then we specify the commands that we want to run.

Finally, we close the loop with the done keyword.

You can also use for to process a series of numbers. For example here is one way to loop through from 1 to 10:

```
#!/bin/bash

for num in {1..10}

do

echo ${num}

done
```

13.6.2 While loops

The structure of a while loop is quite similar to the for loop:

```
while [ your_condition ]

do

 your commands

  done
```

example of a while loop:

```
#!/bin/bash

counter=1

while [[ $counter -le 10 ]]

do

echo $counter

 ((counter++))

 done
```

First, we specified a counter variable and set it to 1, then inside the loop, we added counter by using this statement here: ((counter++)). That way, we make sure that the loop will run 10 times only and would not run forever. The loop will complete as soon as the counter becomes 10, as this is what we've set as the condition: while [[$counter -le 10]].

13.6.3 Until Loops

The difference between until and while loops is that the until loop will run the commands within the loop until the condition becomes true.

Structure:

```
until [[ your_condition ]]
do
    your_commands
done
```

Example:

```
#!/bin/bash
count=1
 until [[ $count -gt 10 ]]
do
echo $count    ((count++))
done
```

13.7 Continue and Break

As with other languages, you can use continue and break with your bash scripts as well:

- ✓ **Continue** tells your bash script to stop the current iteration of the loop and start the next iteration.

The syntax of the continue statement is as follows:

```
continue [n]
```

The [n] argument is optional and can be greater than or equal to 1. When [n] is given, the n-th enclosing loop is resumed. continue 1 is equivalent to continue.

```
#!/bin/bash
for i in 1 2 3 4 5
 do
 if [[ $i –eq 2 ]]
 then
 echo "skipping number 2"
 continue
 fi
  echo "i is equal to $i"
  done
```

We can also use continue command in similar way to break command for controlling multiple loops.

- ✓ **break** tells your bash script to end the loop straight away.

The syntax of the break statement takes the following form:

break [n]

[n] is an optional argument and must be greater than or equal to 1. When [n] is provided, the n-th enclosing loop is exited. break 1 is equivalent to break.

Example:

```
#!/bin/bash
num=1
 while [[ $num –lt 10 ]]
do
  if [[ $num –eq 5 ]]
  then
     break
    fi
   ((num++))
 Done
  echo "Loop completed"
```

We can also use break command with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break 2.

Example:

```
#!/bin/bash
for (( a = 1; a < 10; a++ ))
do
  echo "outer loop: $a"
  for (( b = 1; b < 100; b++ ))
   do
    if [[ $b –gt 5 ]]
  then
  break 2
      fi
  echo "Inner loop: $b "
  done
   done
```

The bash script will begin with a=1 & will move to inner loop and when it reaches b=5, it will break the outer loop. We can use break only instead of break 2, to break inner loop & see how it affects the output

13.8 Exception Handling

Exception handling provides a way to respond to errors during program execution.

- **try-except (Python):**

```
try:
    # Code that may raise an exception
except ExceptionType as e:
    # Code to execute if an exception occurs
```

- **try-catch (Java/C++):**

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to execute if an exception occurs
}
```

13.9 Arithmetic in shell

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by using the ((compound command, the let builtin, or the -i option to the declare builtin.

Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

***id*++ *id*--**

variable post-increment and post-decrement

++*id* --*id*

variable pre-increment and pre-decrement

- +

unary minus and plus

! ~

logical and bitwise negation

******

exponentiation

*** / %**

multiplication, division, remainder

+ -

addition, subtraction

<< >>

left and right bitwise shifts

<= >= < >

comparison

== !=

equality and inequality

&

bitwise AND

^

bitwise exclusive OR

|

bitwise OR

&&

logical AND

||

logical OR

expr ? expr : expr

conditional operator

= *= /= %= += -= <<= >>= &= ^= |=

assignment

expr1 , expr2

Comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the integer attribute using 'declare -i' is assigned a value. A null value evaluates to 0. A shell variable need not have its integer attribute turned on to be used in an expression.

Integer constants follow the C language definition, without suffixes or character constants. Constants with a leading 0 are interpreted as octal numbers. A leading '0x' or '0X' denotes hexadecimal. Otherwise, numbers take the form [*base#*]*n*, where the optional *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base#* is omitted, then base 10 is used. When specifying *n*, if a non-digit is required, the digits greater than 9 are represented by the lowercase letters, the uppercase letters, '@', and '_', in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

13.10 Arrays

Arrays are important concepts in programming or scripting. Arrays allow us to store and retrieve elements in a list form which can be used for certain tasks. In bash, we also have arrays that help us in creating scripts in the command line for storing data in a list format. we will understand the basics of arrays in bash scripting.

Creating Arrays

To create a basic array in a bash script, we can use the declare **-a** command followed by the name of the array variable you would like to give.

```
#!/bin/usr/env bash


declare -a sport=(
[0]=football
[1]=cricket
[2]=hockey
[3]=basketball
)
```

OR

```
#!/bin/usr/env bash
sport[0]=football
sport[1]=cricket
sport[2]=hockey
sport[3]=basketball
```

The value of the elements can be any integer or strings or any other form of data as desired. We can see that the array is declared in a bash script in two ways the former seems more convenient and less hectic to declare. If we want to declare the array in one go, the former is the optimum choice but if the elements are to be added in bits and pieces the latter is a good choice.

Printing the Arrays

After declaring the array, if we wanted to display all the elements in the array we can use the @ symbol.

```
#!/bin/usr/env bash


declare -a sport=(
[0]=football
[1]=cricket
[2]=hockey
[3]=basketball
)
```

```
echo "${sport[@]}"
```

13.11 File and String tests

In Unix-like systems and scripting languages, file and string tests are commonly used to perform checks and validations. These tests are crucial for handling files and strings effectively within scripts. Here's a guide to various file and string tests, particularly in Bash scripting, but also covering some general concepts.

- **File Tests:** Check for existence, type, permissions, size, and more of files.
- **String Tests:** Check if strings are empty, equal, or if one string is less or greater than another.

These tests are fundamental for managing files and manipulating strings in shell scripts, ensuring that scripts behave correctly under various conditions

File Tests in Bash

File tests in Bash are typically done using [ ] or [[ ]] constructs along with test operators. These operators help you check various file attributes and conditions.

File Test Operators

1. **-e**: Checks if a file exists.

   ```
   if [ -e "filename" ]; then
      echo "File exists."
   fi
   ```

2. **-f**: Checks if a file is a regular file (not a directory).

   ```
   if [ -f "filename" ]; then
      echo "Regular file exists."
   fi
   ```

3. **-d**: Checks if a file is a directory.

   ```
   if [ -d "directoryname" ]; then
      echo "Directory exists."
   fi
   ```

4. **-r**: Checks if a file is readable.

   ```
   if [ -r "filename" ]; then
      echo "File is readable."
   fi
   ```

5. **-w**: Checks if a file is writable.

   ```
   if [ -w "filename" ]; then
      echo "File is writable."
   fi
   ```

6. **-x**: Checks if a file is executable.

```
if [ -x "filename" ]; then
   echo "File is executable."
fi
```

7. **-s**: Checks if a file is not empty (i.e., its size is greater than zero).

```
if [ -s "filename" ]; then
   echo "File is not empty."
fi
```

8. **-L**: Checks if a file is a symbolic link.

```
if [ -L "filename" ]; then
   echo "File is a symbolic link."
fi
```

9. **-nt**: Checks if a file is newer than another file.

```
if [ "file1" -nt "file2" ]; then
   echo "file1 is newer than file2."
fi
```

10. **-ot**: Checks if a file is older than another file.

```
if [ "file1" -ot "file2" ]; then
   echo "file1 is older than file2."
fi
```

String Tests in Bash

String tests in Bash allow you to perform operations and comparisons on string values.

String Test Operators

1. **-z**: Checks if a string is empty.

```
if [ -z "$string" ]; then
   echo "String is empty."
fi
```

2. **-n**: Checks if a string is not empty.

```
if [ -n "$string" ]; then
   echo "String is not empty."
fi
```

3. **=**: Checks if two strings are equal.

```
if [ "$string1" = "$string2" ]; then
   echo "Strings are equal."
fi
```

4. **!=**: Checks if two strings are not equal.

```
if [ "$string1" != "$string2" ]; then
   echo "Strings are not equal."
```

```
fi
```

5. <: Checks if one string is less than another (lexicographically).

```
if [[ "$string1" < "$string2" ]]; then
   echo "String1 is less than String2."
fi
```

6. >: Checks if one string is greater than another (lexicographically).

```
if [[ "$string1" > "$string2" ]]; then
   echo "String1 is greater than String2."
fi
```

7. **${#string}**: Returns the length of a string.

```
length=${#string}
echo "Length of the string is $length."
```

Examples

1. **Check if a File Exists and Is Readable:**

```
if [ -e "myfile.txt" ] && [ -r "myfile.txt" ]; then
   echo "File exists and is readable."
fi
```

2. **Compare Two Strings:**

```
string1="hello"
string2="world"

if [ "$string1" = "$string2" ]; then
   echo "Strings are equal."
else
   echo "Strings are not equal."
fi
```

3. **Check if a String Contains a Substring:**

```
main_string="Hello, world!"
substring="world"

if [[ "$main_string" == *"$substring"* ]]; then
   echo "Substring found."
else
   echo "Substring not found."
fi
```

4. **Find Files in a Directory with a Specific Extension:**

```
for file in *.txt; do
   if [ -f "$file" ]; then
```

```
        echo "Found text file: $file"
    fi
done
```

5. **Check if a Directory Exists:**

```
if [ -d "/path/to/directory" ]; then
    echo "Directory exists."
else
    echo "Directory does not exist."
fi
```

13.12 SUMMARY

In this unit, we have learn about the basic concept of Control Flow Statements-Decision, loops and case statements, Arithmetic in Shell Script, Array, File and String Tests in Bash scripting

Check your progress:

- **What is the basic syntax of a bash if-else statement?**

……
……
……
……

- ✓ **What are some common conditional statements frequently used in Bash scripting?**

……
……
……
……

- ✓ **Discuss the about Arithmetic in Shell Script**

……
……
……
……

✓ **Discuss about File and String Tests in Bash scripting**

……
……
……
……

Further Reading:

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education
- ✓ https://www.geeksforgeeks.org/
- ✓ https://www.tutorialspoint.com

Unit-14 : Gawk Programming

Structure

14.1 Introduction

Gawk is the GNU version of the commonly available UNIX **awk** program, another popular stream editor. Since the **awk** program is often just a link to **gawk**, we will refer to it as **awk**.

The basic function of **awk** is to search files for lines or other text units containing one or more patterns. When a line matches one of the patterns, special actions are performed on that line.

Programs in **awk** are different from programs in most other languages, because **awk** programs are "data-driven": you describe the data you want to work with and then what to do when you find it. Most other languages are "procedural." You have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, **awk** programs are often refreshingly easy to read and write.

Objectives:

After studying this unit ,the learner would be able to know about the following:-

- Basic overview of Gawk programming
- What are the variables in Gawk programming
- standard options,

- operators,
- variable and array assignment,
- escape sequences,
- patterns and procedures
- , functions,
- file inclusion,
- output redirections
- printf formats.

14.2 Gawk Command Syntax

The basic **gawk** syntax looks like this:

gawk[option] [action/filter] input file

The command cannot be run without any arguments. The options are not mandatory, but for **gawk** to produce output, at least one action should be assigned. Actions and filters are different subcommands and selection criteria that enable **gawk** to manipulate data from the input file.

14.3 gawk Options

The **gawk** command is a versatile tool thanks to its numerous arguments. With **gawk** being the GNU implementation of **awk**, long, GNU-style options are available. Each long option has a corresponding short one.

Common options are presented below:

Option	Description
-f program-file, --file program-file	Reads commands from a file, which serves as a script, instead of the first argument in the terminal.
-F fs, --field-separator fs	Uses the predefined variable **fs** as the input field separator.
-v var=val, --assign var=val	Assigns a value to the variable before executing a script.
-b, --characters-as-bytes	Treats all data as single-byte characters.
-c, --traditional	Executes **gawk** in compatibility mode.
-C, --copyright	Displays the GNU Copyright message.
-d[file], --dump-variables[=file]	Shows a list of variables, their types, and values.

Option	Description
-e program-text, --source program-text	Allows the mixing of library functions and source code.
-E file, --exec file	Turns off terminal variable assignments.
-L [value], --lint[=value]	Prints warning messages about code not portable to other AWK implementations.
-S, --sandbox	Runs **gawk** in sandbox mode.

14.4 Built-in Variables

The **gawk** command offers several built-in variables used to store and add value to the command. Variables are manipulated from the terminal and only affect the program when a user assigns value to them. Some important **gawk** built-in variables are:

Variable	Description
ARGC	Shows the number of terminal arguments.
ARGIND	Displays the ARGV file index.
ARGV	Presents an array of terminal arguments.
ERRNO	Contains strings describing a system error.
FIELDWIDTHS	Displays white-space separated list of field widths.
FILENAME	Prints the input file name.
FNR	Shows input record number.
FS	Represents the input field separator.
IGNORECASE	Turns case-sensitive search on or off.
NF	Prints the input file field count.

Variable	Description
NR	Prints the current file line count.
OFS	Displays the output field separator.
ORS	Shows the output record separator.
RS	Prints the input record separator.
RSTART	Represents the index of the first matched character.
RLENGTH	Represents the matched string length.

14.5 Operators

Gwak is a lesser-known programming language, so detailed information on it might be a bit sparse. If you're referring to Gwak, a unique language created for educational purposes or specific use cases, here are some typical operators and constructs you might encounter:

Basic Operators

1. **Arithmetic Operators**:
    - + : Addition
    - - : Subtraction
    - * : Multiplication
    - / : Division
    - % : Modulus (remainder)
2. **Comparison Operators**:
    - == : Equal to
    - != : Not equal to
    - > : Greater than
    - < : Less than
    - >= : Greater than or equal to
    - <= : Less than or equal to
3. **Logical Operators**:
    - && : Logical AND
    - || : Logical OR
    - ! : Logical NOT

4. **Assignment Operators**:
    - = : Assignment
    - += : Add and assign
    - -= : Subtract and assign
    - *= : Multiply and assign
    - /= : Divide and assign
    - %= : Modulus and assign

Control Flow Operators

1. **Conditional Statements**:
    - if, else if, else
2. **Loops**:
    - for : Loop with a defined number of iterations
    - while : Loop while a condition is true
    - do-while : Loop at least once, then repeat while a condition is true

Special Constructs

1. **Function Calls**:
    - Usually involve calling defined functions with parameters.
2. **Comments**:
    - // for single-line comments
    - /* ... */ for multi-line comments

14.6 Variable and array assignment

In gawk, which stands for GNU awk, variables are used for storing data that can be manipulated and used in various operations within your awk programs. Here's a brief overview of how variables work in gawk:

Declaring Variables

Variables in gawk don't need explicit declarations. You can start using them directly. If you assign a value to a variable, it is automatically created.

awk

Copy code

x = 10

y = "Hello, world!"

Variable Types

- **Strings**: Variables can hold string values.

  awk

Copy code

name = "Alice"

- **Numbers**: Variables can hold numeric values (integers or floating-point).

  awk

  Copy code

  age = 30

  height = 5.9

- **Arrays**: gawk supports associative arrays, which can be used to store values with string keys.

  awk

  Copy code

  fruits["apple"] = 1

  fruits["banana"] = 2

Using Variables

You can use variables in expressions, print statements, and various awk functions.

awk

Copy code

x = 5

y = 10

sum = x + y

print sum

Built-in Variables

gawk provides several built-in variables that have special meanings:

- **$0**: The entire current record (usually a line of input).
- **$1, $2, ...**: Fields in the current record (if the record is split into fields).
- **NR**: The number of records processed so far.
- **NF**: The number of fields in the current record.
- **FS**: The input field separator.
- **OFS**: The output field separator.
- **RS**: The input record separator.
- **ORS**: The output record separator.

Example Usage

Here's a simple example to demonstrate variable usage in a gawk script:

awk

Copy code

```
BEGIN {
   FS = ","          # Set the input field separator to a comma
   OFS = ";"         # Set the output field separator to a semicolon
}

{
   total = $1 + $2   # Add the first and second fields
   print $1, $2, total
}
```

In this script:

- FS is used to specify that fields in the input are separated by commas.
- OFS is used to specify that fields in the output should be separated by semicolons.
- total is a variable that stores the result of adding the first and second fields of each record.

Modifying Variables

You can change the value of a variable at any point in your script:

awk

Copy code

```
x = 5
print x       # Outputs 5
x = x + 10
print x       # Outputs 15
```

Arrays and Variables

Arrays in gawk are associative, meaning that the indices can be strings or numbers.

awk

Copy code

```
count["apple"] = 3
count["banana"] = 5
print count["apple"]  # Outputs 3
print count["banana"] # Outputs 5
```

Variables in gawk are flexible and powerful, allowing for a wide range of operations and manipulations within your awk scripts.

14.7 Escape sequence

Escape sequences are special character sequences used in programming to represent characters that are difficult or impossible to include directly in strings. They are often used to represent control characters, formatting, or special symbols. Escape sequences are a powerful feature in gawk and other

programming languages, allowing for more control over the formatting and content of strings. They help include special characters that cannot be directly typed into strings or need to be represented in a specific way.

In gawk, which is a version of the awk programming language, escape sequences can be used within string literals to include special characters. The common escape sequences used in gawk:

Common Escape Sequences in gawk

1. **\n**: Newline character.
    - Inserts a line break.
    - Example: print "Hello\nWorld"

outputs:

Hello

World

2. **\t**: Horizontal tab.
    - Inserts a tab space.
    - Example: print "Column1\tColumn2"

      Outputs:

      Column1 Column2
3. **\r**: Carriage return.
    - Moves the cursor to the beginning of the line.
    - Example: print "123\rABC"

      Outputs:

      ABC
4. \\: Backslash character.
    - Inserts a literal backslash.
    - Example: print "Backslash: \\"

      outputs:

      makefile

      Backslash: \
5. \": Double quote character.
    - Inserts a literal double quote within a string.
    - Example: print "He said, \"Hello!\""

      outputs:

      He said, "Hello!"
6. **\b**: Backspace.
    - Moves the cursor one position back (though the effect might not be visible in all environments).

- Example: print "123\b4" might
  output:
  124

7. **\f**: Form feed.
   - Moves to the next page or section (effects can vary by environment).
   - Example: print "Page1\fPage2" may have a page break in some environments.
8. **\a**: Alert (bell).
   - Produces a bell sound or alert (not always supported in all environments).
   - Example: print "\a" may produce a beep sound.
9. **\v**: Vertical tab.
   - Moves to the next vertical tab stop (rarely used).
   - Example: print "Line1\vLine2" might have vertical spacing in some environments.

Examples in gawk

Here are a few examples demonstrating the use of escape sequences in gawk:

```
# Print a string with a newline
BEGIN {
  print "Hello\nWorld"
}

# Print a string with tabs
BEGIN {
  print "Name\tAge\tLocation"
  print "Alice\t30\tNew York"
  print "Bob\t25\tSan Francisco"
}

# Include a backslash and double quote in the string
BEGIN {
  print "A backslash: \\"
  print "A quote: \""
}
```

14.8 Patterns and procedures in gawk

In gawk, patterns and procedures (often referred to as actions) are key concepts used to process and manipulate text data. Understanding how to use them effectively is crucial for writing powerful awk scripts.

- **Patterns**: Define which lines to process, using regular expressions, boolean conditions, or special patterns like BEGIN and END.
- **Procedures (Actions)**: Define what to do with lines that match the patterns, enclosed in curly braces {}.

By combining patterns and procedures, gawk allows for powerful text processing and data manipulation.

Patterns

In gawk, patterns specify which lines of input should be processed. A pattern is a condition that is checked against each line of input data, and if the pattern matches, the associated action (procedure) is executed.

Patterns can be:

1. **Regular Expressions**:

Patterns can be regular expressions used to match specific text patterns in each line.

Example: /^Error/ matches lines that start with the word "Error".

```
/^Error/ {
  print "Error found:", $0
}
```

2. **Boolean Expressions**:
    - These can be any boolean condition evaluated against each line. For example, $1 > 100 matches lines where the first field is greater than 100.

```
$1 > 100 {
  print $0
}
```

3. **Patterns with Comparison Operators**:
    - You can use comparison operators to match lines based on specific conditions.

```
NR == 5 {
  print "Fifth record:", $0
}
```

4. **BEGIN and END Patterns**:
    - BEGIN and END are special patterns that are used for actions before processing any input lines or after processing all input lines, respectively.

```
BEGIN {
  print "Start processing"
}


END {
```

```
    print "End processing"
}
```

Procedures (Actions)

Procedures in gawk are blocks of code that are executed when a pattern matches. The action block is enclosed in curly braces {} and contains commands to be executed.

Here's how you typically structure a gawk script:

```
pattern {
  action
}
```

If no pattern is specified, the action applies to all lines of input.

Examples

1. **Basic Example**:

   ```
   /pattern/ {
     print $0
   }
   ```

   This script prints every line that matches "pattern".

2. **Multiple Patterns and Actions**:

   ```
   /^Error/ {
     print "Error line:", $0
   }

   /^[0-9]/ {
     print "Number line:", $0
   }
   ```

   This script prints lines starting with "Error" with a specific message and lines starting with digits with a different message.

3. **Using BEGIN and END**:

   ```
   BEGIN {
     print "Processing started"
   }

   /pattern/ {
     print "Matched line:", $0
   }
   ```

```
END {
   print "Processing finished"
}
```

This script prints messages at the beginning and end of processing, and also processes lines matching the pattern.

4. **Field Processing**:

```
$1 == "John" {
   print "John's record:", $0
}
```

This script processes lines where the first field is "John" and prints the entire line.

5. **Complex Example with Calculations**:

```
BEGIN {
   sum = 0
}

{
   sum += $2
}

END {
   print "Total sum of second field:", sum
}
```

This script sums up the values in the second field of each line and prints the total at the end.

14.9 Functions

In gawk, functions are essential for structuring and modularizing your code. They allow you to define reusable blocks of code, making your awk scripts more organized and easier to maintain. Functions in gawk can be built-in or user-defined.

- **Built-in Functions**: Provided by gawk for common tasks, such as string manipulation, mathematical calculations, and file handling.
- **User-Defined Functions**: Created by the user to encapsulate reusable code logic. Defined with the function keyword and can be called with parameters.

By using both built-in and user-defined functions, you can write more modular and maintainable gawk scripts.

Built-in Functions

gawk provides a range of built-in functions that perform common tasks. Here are some of the most commonly used built-in functions:

1. **String Functions**:
   - **length([string])**: Returns the length of the string or the length of the current record if no argument is provided.

     { print length($0) }
   - **substr(string, start, length)**: Returns a substring of string starting from start and with the specified length.

     { print substr($0, 1, 5) }
   - **index(string, substring)**: Returns the position of substring in string or 0 if not found.

     { print index($0, "pattern") }
   - **tolower(string)**: Converts string to lowercase.

     { print tolower($0) }
   - **toupper(string)**: Converts string to uppercase.

     { print toupper($0) }
2. **Numeric Functions**:
   - **sqrt(x)**: Returns the square root of x.

     {

     print sqrt($1)

     }
   - **exp(x)**: Returns e raised to the power of x.

     { print exp($1) }
   - **log(x)**: Returns the natural logarithm of x.

     { print log($1) }
   - **sin(x), cos(x), atan2(y, x)**: Trigonometric functions.

     {

     print sin($1), cos($1)

     }
3. **Date and Time Functions**:
   - **strftime(format, time)**: Formats the time according to the format string. time is optional; if omitted, the current time is used.

     { print strftime("%Y-%m-%d %H:%M:%S") }
4. **File and I/O Functions**:
   - **getline [var]**: Reads the next record into var or into $0 if var is not specified.

     getline x

print x

- **close(file)**: Closes the specified file.

  close("file.txt")

User-Defined Functions

You can also define your own functions in gawk. User-defined functions allow you to encapsulate logic and reuse it throughout your script.

Defining Functions: A function is defined with the function keyword followed by the function name, a list of parameters, and the function body.

```
function myFunction(param1, param2) {
   # Function body
   return result
}
```

Example:

```
# Define a function to calculate the square of a number
function square(x) {
   return x * x
}

# Use the function in a pattern-action statement
{
   print "The square of", $1, "is", square($1)
}
```

In this example, square is a user-defined function that calculates the square of a number. This function is then used to process each line of input.

Example Script with User-Defined Functions

Here's a more comprehensive example of a gawk script that includes user-defined functions:

```
# Define a function to calculate the area of a rectangle
function rectangleArea(width, height) {
   return width * height
}

# Define a function to calculate the area of a circle
function circleArea(radius) {
   return 3.14159 * radius * radius
}
```

```
BEGIN {
    print "Calculating areas:"
    width = 10
    height = 5
    radius = 7

    print "Area of rectangle (10x5):", rectangleArea(width, height)
    print "Area of circle with radius 7:", circleArea(radius)
}
```

In this script:

- rectangleArea calculates the area of a rectangle.
- circleArea calculates the area of a circle.
- The BEGIN block initializes values and prints the results of the calculations.

14.10 FILE INCLUSION

In gawk, file inclusion allows you to split your awk scripts into multiple files, which can be useful for organizing complex scripts or reusing common code. gawk provides a way to include external files using the -f option and the include statement.

- **Using -f Option**: Allows you to include external files containing awk code by specifying the file with -f.
- **Using @include Directive**: Directly includes external files within your awk script, supported in gawk 4.2.0 and later.
- **Multiple -f Options**: Allows specifying multiple files to be included in sequence.
- **Dynamic Inclusion**: Advanced technique using getline and eval for dynamic file inclusion, though less common.

These methods help you manage large awk scripts by breaking them into smaller, more manageable pieces and reusing code effectively.

1. Using the -f Option

The -f option allows you to specify a file containing awk code that should be executed. This is useful for separating your awk script into a main file and additional files containing reusable functions or code.

Example:

Suppose you have a main script file called main.awk and a separate file with functions called functions.awk.

functions.awk:

```
# Function to calculate the square of a number
function square(x) {
```

```
    return x * x
}

# Function to print a greeting
function greet(name) {
    print "Hello, " name "!"
}
```

main.awk:

```
# Include the functions from the file
@include "functions.awk"

BEGIN {
    name = "Alice"
    number = 4

    greet(name)         # Call the function from functions.awk
    print "The square of", number, "is", square(number)
}
```

To run the script, use the -f option to include the main.awk file, which will also include functions.awk:

```
gawk -f main.awk
```

2. Using @include Directive

Starting from gawk version 4.2.0, you can use the @include directive within gawk scripts to include external files. This is similar to including files in other programming languages.

Example:

main.awk:

```
@include "functions.awk"

BEGIN {
    name = "Bob"
    number = 7

    greet(name)
    print "The square of", number, "is", square(number)
}
```

Here, functions.awk is included directly in main.awk using the @include directive.

3. Using awk Scripts from Command Line

If you have multiple awk scripts you want to run in sequence, you can specify multiple -f options. This way, you can include several files without manually editing your main script.

Example:

gawk -f functions.awk -f main.awk

In this command, functions.awk will be loaded first, followed by main.awk.

4. Including Files Dynamically

If you need to include files dynamically, you can use the getline function to read the content of a file and execute it as awk code. This approach is more advanced and less common, but it can be useful in certain scenarios.

Example:

dynamic.awk:

```
BEGIN {
   while ((getline line < "functions.awk") > 0) {
      eval(line)  # Executes the line as `awk` code
   }
   close("functions.awk")

   name = "Charlie"
   number = 9

   greet(name)
   print "The square of", number, "is", square(number)
}
```

14.11 Output redirection

Output redirection in gawk allows you to control where the output of your awk script is sent. By default, gawk prints output to the standard output (usually the terminal). However, you can redirect this output to files, other commands, or even append it to existing files.

- **Shell Redirection**: Use > and >> to redirect output to files.
- **print Statement**: Sends output to standard output, which can be redirected by the shell.
- **printf Function**: Provides formatted output control.
- **File Output within gawk**: Use print with a file handle (> file) for output redirection within the script.
- **System Commands**: Use system() to run shell commands, which can include redirection.

By utilizing these methods, you can effectively manage where and how your gawk script outputs its data.

1. Redirecting Output to a File

You can redirect the output of your gawk script to a file using the shell's redirection operators. This is done outside of the gawk script, in the shell command line.

Example:

gawk '{ print $0 }' input.txt > output.txt

In this example, gawk reads from input.txt and writes the output to output.txt. The > operator overwrites the file if it already exists.

To append the output to an existing file instead of overwriting it, use the >> operator:

gawk '{ print $0 }' input.txt >> output.txt

2. Using gawk's print Statement

Within a gawk script, you can use the print statement to send output to the standard output, which is usually the terminal or the file specified by redirection.

Example:

```
# sample.awk
{
   print "Name:", $1, "Age:", $2
}
```

To execute this awk script and redirect its output to a file:

gawk -f sample.awk input.txt > output.txt

3. Using print and printf to Control Output

You can use print and printf within gawk scripts to format and control the output:

- **print Statement**: Outputs data with a space between each item.

  ```
  # Prints a line with space-separated fields
  { print $1, $2 }
  ```

- **printf Function**: Allows for more precise formatting of output, similar to the printf function in C.

  ```
  # Prints a formatted string
  { printf "Name: %-10s Age: %d\n", $1, $2 }
  ```

4. Using gawk's print to Direct Output to a File

You can also use gawk's built-in print function to write to a file within the script. This method uses the > operator to specify the file to which output should be redirected.

Example:

```
# output.awk
BEGIN {
```

```
    file = "output.txt"
    print "Name", "Age" > file
}

{
    print $1, $2 > file
}
```

In this script:

- The BEGIN block opens output.txt and prints headers.
- Each subsequent line appends output to output.txt.

5. Redirecting Output from Multiple Files

To redirect output from multiple files or commands, you can use the shell's redirection features:

Example:

```
gawk -f script.awk file1.txt file2.txt > combined_output.txt
```

This command processes file1.txt and file2.txt with script.awk and writes the combined output to combined_output.txt.

6. Redirection Inside gawk with system()

For more complex scenarios where you need to invoke system commands from within gawk, use the system() function. While this doesn't directly handle redirection, it allows for command execution which can include redirection.

Example:

```
# example.awk
BEGIN {
    cmd = "echo 'Hello from gawk' > output.txt"
    system(cmd)
}
```

14.12 Printf formats

In gawk, the printf function provides powerful and flexible formatting options for output. It allows you to control the layout, precision, and alignment of your output, similar to the printf function in the C programming language.

- **%s**: String
- **%d**: Decimal integer
- **%f**: Floating-point number
- **%.nf**: Floating-point with n decimal places
- **%x, %X**: Hexadecimal integer (lowercase/uppercase)
- **%o**: Octal integer

- **%e**: Scientific notation
- **%g**: General format
- **Field Width**: Specify minimum number of characters
- **Precision**: For floating-point numbers, specify decimal places
- **Alignment**: Use - for left alignment

Using printf effectively allows for precise control over the output format in gawk, making it a powerful tool for generating well-formatted reports and data.

Basic Syntax

The basic syntax for printf in gawk is:

printf format, argument1, argument2, ...

- **format**: A format string that specifies how the subsequent arguments should be formatted.
- **argument1, argument2, ...**: Values to be formatted and output.

Common Format Specifiers

Here are some common format specifiers you can use with printf:

1. **%s**: String
    - Example: printf "%s\n", "Hello"
      Output: Hello
2. **%d**: Decimal integer
    - Example: printf "%d\n", 123
      Output: 123
3. **%f**: Floating-point number
    - Example: printf "%f\n", 3.14159
    - Output: 3.141590
4. **%.nf**: Floating-point number with n decimal places
    - Example: printf "%.2f\n", 3.14159
    - Output: 3.14
5. **%x**: Hexadecimal integer (lowercase)
    - Example: printf "%x\n", 255
    - Output: ff

6. **%X**: Hexadecimal integer (uppercase)
    - Example: printf "%X\n", 255
    - Output: FF
7. **%o**: Octal integer
    - Example: printf "%o\n", 8
    - Output: 10
8. **%e**: Scientific notation

    Example: printf "%e\n", 1234.5678

    Output: 1.234568e+03

9. **%g**: General format (either fixed-point or scientific notation)

    Example: printf "%g\n", 1234.5678

    Output: 1234.57

Field Width and Precision

You can specify the minimum field width and precision for the output using format specifiers.

- **Field Width**: Specifies the minimum number of characters to be printed. If the value is shorter, it will be padded with spaces.
- **Precision**: For floating-point numbers, it specifies the number of digits after the decimal point.

Examples:

- **Field Width**:
  printf "|%10s|%10d|\n", "Name", 123
  Output: | Name| 123|
- **Precision for Floating-Point Numbers**:
  printf "%.3f\n", 3.14159
  Output: 3.142

- **Combined Width and Precision**:
  printf "|%10.2f|\n", 3.14159
  Output: | 3.14|

Left and Right Alignment

By default, printf right-aligns text. To left-align text, use a - sign in the format specifier.

Examples:

- **Right-Alignment (Default)**:
  printf "|%10s|\n", "Text"
  Output: | Text|

- **Left-Alignment**:
  printf "|%-10s|\n", "Text"
  Output: |Text |

Multiple Arguments and Format Specifiers

You can use multiple format specifiers and arguments in a single printf call.

Example:

printf "Name: %-10s Age: %3d Height: %.2f\n", "Alice", 30, 5.678

Output: Name: Alice Age: 30 Height: 5.68

14.13 SUMMARY

In this unit, we have learn about the basic concept of gwak programming , command line syntax, standard options, Built in variables, operators, variable and array assignment, escape sequences, patterns and procedures, functions, file inclusion, output redirections, printf formats.

14.14 Check your progress

- **What is gwak script?**

……
……
……
……

✓ **What do you mean by file inclusion in gwak script?**

……
……
……
……

✓ **Discuss the about Arithmetic in gwak Script**

……
……
……
……

✓ **Discuss about printf formats in gwak scripting**

……
……
……
……

14.15 Further Reading

- ✓ Unix and shell programming by B.M. Harwani, OXFORD university press
- ✓ UNIX : Concepts and Applications by Sumitabha Das TMH Education

https://www.geeksforgeeks.org/

https://phoenixnap.com/kb/gawk-linux

https://linux.die.net/Bash-Beginners-Guide/chap_06.htmlForm

Notes

Notes

Notes