ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle and Graded as Category–I University by MHRD-UGC]

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

B.C.A.

10154

UNIX & SHELL PROGRAMMING LAB

V - Semester

UNIX & SHELL PROGRAMMING LAB

B.C.A.

ALAGAPPA UNIVERSITY

(Accredited with 'A+' Grade by NAAC (with CGPA: 3.64) in the Third Cycle and Graded as category - I University by MHRD-UGC)
(A State University Established by the Government of Tamilnadu)

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

BACHELOR OF COMPUTER APPLICATIONS

Third Year – Fifth Semester

10154 – UNIX & SHELL PROGRAMMING LAB

Copy Right Reserved For Private Use only

Author :
Dr. K. Shankar
Assistant Professor
Department of Computer Science and Information Technology
Kalasalingam Academy of Research and Education
Anand Nagar, Krishnankoil-626126

"The Copyright shall be vested with Alagappa University"

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Alagappa University, Karaikudi, Tamil Nadu.

Reviewer:
Mr. S. Balasubramanian
Assistant Professor in Computer Science
Directorate of Distance Education
Alagappa University
Karaikudi – 03.

SYLLABUS

BLOCK 1

EX.NO: 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 1960's definition of an operating system is "software that controls the hardware". However, today, due to microcode we need a better definition. The operating system is viewed as the programs that make the hardware useable. In brief, an operating system is the set of programs that controls a computer. An Operating system is software that creates a relation between the User, Software and Hardware. It is an interface between the all. All the computers need basic software known as an Operating System (OS) to function. The OS acts as an interface between the User, Application Programs, Hardware and the System Peripherals. The OS is the first software to be loaded when a computer starts up. The entire application programs are loaded after the OS.

Types of Operating System (Based on No. of user)

- **Single User**: If the single user Operating System is loaded in computer's memory; the computer would be able to handle 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-2000-200**3**

Objective

To understand the UNIX OS and its programming language from the scratch. After going through this section, you should be able to:

• Mention the features of UNIX;
• Recognize, understand and make use of various UNIX commands;
• Gain hands on experience of UNIX commands and shell programs;
• Feel more confident about writing the shell scripts and shell programs;
• Apply the concepts that have been covered in this manual,

• Know the alternate ways of providing the solutions to the given practical exercises and problems.

History of UNIX

Imagine computers as big as houses, even stadiums. While the sizes of those computers posed substantial problems, there was one thing that made this even worse: every computer had a different operating system. Software was always customized to serve a specific purpose, and software for one given system didn't run on another system. Being able to work with one system didn't automatically mean that you could work with another. It was difficult, both for the users and the system administrators. Technologically the world was not quite that advanced, so they had to live with the size for another decade. In 1969, a team of developers in the Bell Labs laboratories started working on a solution for the software problem, to address these compatibility issues. They developed a new operating system, which was

- Simple and elegant.
- Written in the C programming language instead of in assembly code.
- Able to recycle code.

The Bell Labs developers named their project "UNIX." The code recycling features were very important. Until then, all commercially available computer systems were written in a code specifically developed for one system. UNIX on the other hand needed only a small piece of that special code, which is now commonly named the kernel. This kernel is the only piece of code that needs to be adapted for every specific system and forms the base of the UNIX system. The operating system and all other functions were built around this kernel and written in a higher programming language, C.

This language was especially developed for creating the UNIX system. Using this new technique, it was much easier to develop an operating system that could run on many different types of hardware. The software vendors were quick to adapt, since they could sell ten times more software almost effortlessly. Weird new situations came in existence: imagine for instance computers from different vendors communicating in the same network, or users working on different systems without the need for extra education to use another computer. UNIX did a great deal to help users become compatible with different systems. Throughout the next couple of decades, the development of UNIX continued. More things became possible to do and more hardware and software vendors added support for UNIX to their products.

UNIX was initially found only in very large environments with mainframes and minicomputers (note that a PC is a "micro" computer). But smaller computers were being developed, and by the end of the 80s, many people had home computers. By that time, there were several versions of UNIX

available for the PC architecture, but none of them were truly free and more important: they were all terribly slow, so most people ran MS DOS or Windows 3.1 on their home PCs.

Linus Torvalds, a young man studying computer science at the University of Helsinki, used the Minix, and when he felt too constrained by its limitation, he started to code his own UNIX look like operating system. From the start, it was Linus' goal to have a free system that was completely compliant with the original UNIX. That is why he asked for POSIX standards, POSIX still being the standard for UNIX.

In those days plug-and-play wasn't invented yet, but so many people were interested in having a UNIX system of their own, that this was only a small obstacle. New drivers became available for all kinds of new hardware, at a continuously rising speed. Almost as soon as a new piece of hardware became available, someone bought it and submitted it to the Linux test, as the system was gradually being called, releasing more free code for an ever-wider range of hardware. These coders didn't stop at their PCs; every piece of hardware they could find was useful for Linux. Two years after Linus' post, there were 12000 Linux users. The project, popular with hobbyists, grew steadily, all the while staying within the bounds of the POSIX standard. All the features of UNIX were added over the next couple of years, resulting in the mature operating system Linux has become today.

Linux is a full UNIX clone, fit for use on workstations as well as on middle range and high-end servers. Today, a lot of the important players in the hardware and software market each have their team of Linux developers; at your local dealers you can even buy preinstalled Linux systems with official support–though there is still some hardware and software not supported

What Is UNIX?

- UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since.
- It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.
- UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment.
- Knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

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.

Features of UNIX OS

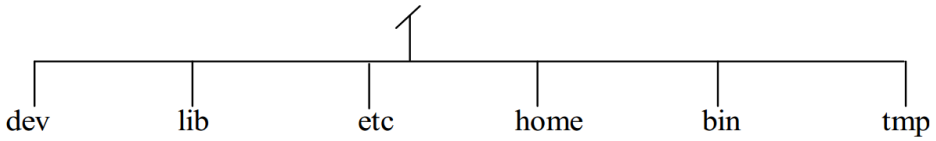
Multitasking

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

- **Multi-user capability**
  This allows several users to use the same computer to perform their tasks.
- **Security**
  Every user has a login name and a password. So, accessing another user's data is impossible without permission

- **Portability**
  UNIX is portable because it is written in a high level language ©. So UXIX can be run on different computers.
- **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
- **Programming facility**
  UNIX is highly programmable; the UNIX shell programming language has all the necessary ingredients like conditional and control structures (Loops) and variables.

Structure of a UNIX file system



/lib	library files
/dev	Contains file that link hardware devices
/tmp	temporary storage of files
/home	user home directories

EX. NO: 2. 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.

The Various Operation performed by the Kernel Listed below:

- 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.

kernel and shell

NOTES

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).

EX. NO: 3 UNIX FILE 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 (typically related) 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):

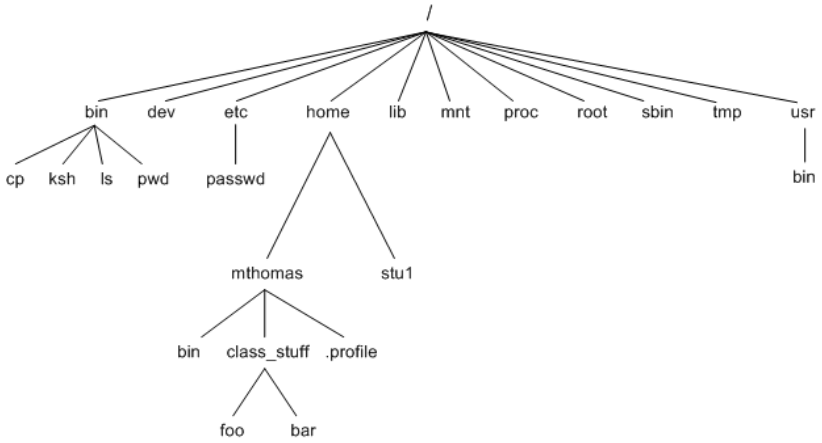
- a file type (i.e. what kind of data is in the file)
- a file name (which may or may not include an extension)
- a physical file size
- a file owner
- file protection/privacy capability
- file time stamp (time and date created/modified)

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 filesystems:

- 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

Common Commands

Touch: Create a new file or update its timestamp.

- **Syntax**: touch [OPTION]…[FILE]
- **Example**: Create empty files called ‘file1’ and ‘file2’
  - $ touch file1 file2

Cat: Concatenate files and print to stdout.

- **Syntax**: cat [OPTION]…[FILE]
- **Example**: Create file1 with entered content
  - $ cat > file1
  - Hello
  - ^D

cp: Copy files

- **Syntax**: cp [OPTION]source destination
- **Example**: Copies the contents from file1 to file2 and contents of file1 is retained
  - $ cp file1 file2

Mv: Move files or rename files

- **Syntax**: mv [OPTION]source destination

- **Example**: Create empty files called ‘file1’ and ‘file2’
  - $ mv file1 file2

rm: Remove files and directories

- **Syntax**: rm [OPTION]…[FILE]
- **Example**: Delete file1
  - $ rm file1

mkdir: Make directory

- **Syntax**: mkdir [OPTION] directory
- **Example**: Create directory called dir1
  - $ mkdir dir1

rmdir: Remove a directory

- **Syntax**: rmdir [OPTION] directory
- **Example**: Create empty files called ‘file1’ and ‘file2’
- $ rmdir dir1

Cd: Change directory

- **Syntax**: cd [OPTION] directory
- **Example**: Change working directory to dir1
- $ cd dir1

pwd: Print the present working directory

- **Syntax**: pwd [OPTION]
- **Example**: Print ‘dir1’ if a current working directory is dir1
- $ pwd

File and Directory Related commands

1. **pwd**
   This command prints the current working directory
2. **ls**
   This command displays the list of files in the current working directory.
   $ls –l Lists the files in the long format
   $ls –t Lists in the order of last modification time
   $ls –d Lists directory instead of contents
   $ls -u Lists in order of last access time
3. **cd**
   This command is used to change from the working directory to any other directory specified.
   $cd directoryname
4. **cd..**
   This command is used to come out of the current working directory.
   $cd.
5. **mkdir**
   This command helps us to make a directory.
   $mkdir directoryname
6. **rmdir**

NOTES

This command is used to remove a directory specified in the command line. It requires the specified directory to be empty before removing it.

$rmdir directoryname

7. cat

This command helps us to list the contents of a file we specify.

$cat [option][file]

cat > filename – This is used to create a new file.

cat >>filename – This is used to append the contents of the file

8. cp

This command helps us to create duplicate copies of ordinary files.

$cp source destination

9. mv

This command is used to move files.

$mv source destination

10. ln

This command is to establish an additional filename for the same ordinary file.

$ln first name second name

11. rm

This command is used to delete one or more files from the directory.

$rm [option] filename

$rm –i Asks the user if he wants to delete the file mentioned.

$rm –r Recursively delete the entire contents of the directory as well as the directory itself.

Process and status information commands

1) who

This command gives the details of who all have logged in to the UNIX system currently.

$ who

2) who am i

This command tells us as to when we had logged in and the system's name for the connection being used.

$who am i

3) date

This command displays the current date in different formats.

+%D	mm/dd/yy	+%w	Day of the week
+%H	Hr-00 to 23	+%a	Abbr.Weekday
+%M	Min-00 to 59	+%h	Abbr.Month

+%S	Sec-00 to 59	+%r	Time in AM/PM
+%T	HH:MM:SS	+%y	Last two digits of the year

4) echo

This command will display the text typed from the keyboard.

$echo

Eg **:** $echo Have a nice day

O/p **:** Have a nice day

Text related commands

1. head

This command displays the initial part of the file. By default it displays first ten lines of the file.

$head [-count] [filename]

2. tail

This command displays the later part of the file. By default it displays last ten lines of the file.

$tail [-count] [filename]

3. wc

This command is used to count the number of lines, words or characters in a file.

$wc [-lwc] filename

4. find

The find command is used to locate files in a directory and in a subdirectory.

The –name option

This lists out the specific files in all directories beginning from the named directory. Wild cards can be used.

The –type option

This option is used to identify whether the name of files specified are ordinary files or directory files. If the name is a directory then use "-type d " and if it is a file then use "-type f".

The –mtime option

This option will allow us to find that file which has been modified before or after a specified time. The various options available are –mtime n(on a particular day),-mtime +n(before a particular day),-mtime –n(after a particular day)

The –exec option

This option is used to execute some commands on the files that are found by the find command.

File Permission commands

1. **chmod**
   Changes the file/directory permission mode: $ chmod 777 file1
   Gives full permission to owner, group and others
   $ chmod o-w file1
   Removes write permission for others.

Useful Commands:

1. **Exit** - Ends your work on the UNIX system.

Shell

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively – enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled.

Common Shells.

- **C-Shell** - csh : The default on teaching systems Good for interactive systems Inferior programmable features
- **Bourne S**hell - bsh or sh - also restricted shell - bsh : Sophisticated pattern matching and file name substitution
- **Korn Shell**: Backwards compatible with Bourne Shell Regular expression
  Substitution emacs editing mode
- **Thomas C-Shell** - tcsh : Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shel

Shell Variables:

Shell variables change during the execution of the program. The C Shell offers a
Command "Set" to assign a value to a variable.

For example:

% set myname= Fred
% set myname = "Fred Bloggs"
% set age=20
A $ sign operator is used to recall the variable values.

For example:

% echo $myname will display Fred Bloggs on the screen
A @ sign can be used to assign the integer constant values.

For example:

%@myage=20
%@age1=10

```
%@age2=20
%@age=$age1+$age2
%echo $age
List variables
% set programming_languages= (C LISP)
% echo $programming _languages
```

C LISP

```
% set files=*.*
% set colors=(red blue green)
% echo $colors[2]
blue
% set colors=($colors yellow)/add to list
```

Local variables Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

Example

```
variable_name=value
name="John Doe"
x=5
```

Global variables Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

Example

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Operating System Lab Manual CS 2254

@www.getitcse.tk Page 17

Extracting values from variables To extract the value from variables, a dollar sign is used.

Example

```
echo $variable_name
echo $name
echo $PATH
```

Rules :

1. A variable name is any combination of alphabets, digits and an underscore („-„);
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variables names should be of any reasonable length.

5. Variables name are case sensitive. That is, Name, NAME, name, Name, are all different variables.

MORE ABOUT FILES

NOTES

All files in the UNIX file system can be loosely categorized into 3 types, specifically:

1. Ordinary files
2. Directory files

The first type of file listed above is an ordinary file, that is, a file with no "special-ness". Ordinary files are comprised of streams of data (bytes) stored on some physical device. Examples of ordinary files include simple text files, application data files, files containing high-level source code, executable text files, and binary image files. Note that unlike some other OS implementations, files do not have to be binary Images to be executable (more on this to come).

The second type of file listed above is a special file called a directory (please don't call it a folder?). Directory files act as a container for other files, of any category. Thus, we can have a directory file contained within a directory file (this is commonly referred to as a subdirectory). Directory files don't contain data in the user sense of data, they merely contain references to the files contained within them.

It is perhaps noteworthy at this point to mention that any "file" that has files directly below (contained within) it in the hierarchy mus**t** be a directory, and any "file" that does not have files below it in the hierarchy can be an ordinary file, or a directory, albeit empty.

Creating Files with cat

There are many ways of creating a file
One of the simplest is with the cat command:

$ cat > shopping_list
cucumber
bread
yoghurts
fish fingers

Note the greater-than sign (>) — this is necessary to create the file
The text typed is written to a file with the specified name
Press Ctrl+D after a line-break to denote the end of the file
The next shell prompt is displayed
ls demonstrates the existence of the new file

Displaying Files' Contents with cat

There are many ways of viewing the contents of a file

One of the simplest is with the cat command:
$ cat shopping_list
cucumber
bread
yoghurts
fish fingers
Note that no greater-than sign is used
The text in the file is displayed immediately:
Starting on the line after the command
Before the next shell prompt

Directories of UNIX System

Linux "folders" are called directories. The top-level, root directory is called /. Your home directory is /home/username. From anywhere you can get back there by typing simply cd The short-hand name for the directory you happen to be in at any time is called "." and the directory in which the current directory resides is called "..". Typing "cd." will move you to the next higher level directory. Several useful commands for directories are listed below.

Command Function Examples

Cd -	Change directory cd, cd ..,	cd /home/catyp
Pwd-	Print working directory	pwd
Mkdir-	Make a new subdirectory	mkdir new directory
Rmdir-	Remove a directory	rmdir empty directory
ls -	List files in a directory	ls, ls –l

Files:

Files reside in directories. Use the ls command (or ls -l for more information) to see all the files in a directory. Useful commands for manipulating files include:

Command Function Examples

Mv	Rename (move) a file	mv oldname newname
cp	Copy a file	cp ol dname newname
		cp oldname dirname/
rm	Delete (remove) a file	rm filename
rm	file1 file2 file3	rm -r dirname

cat Output the contents of a file cat filename to the screen
file identify the type of file file filename
head Display the first few lines head filename of a text file.
tail Display the last few lines tail filename of a text file.
chmod Change access permissions on files chmod mode filename
ln Creates symbolic link ln -s targetfile linkname

A directory is a collection of files and/or other directories. Because a directory can contain other directories, we get a directory hierarchy. The 'top level' of the hierarchy is the root directory. Files and directories can be named by a path. Shows programs how to find their way to the file .The

root directory is referred to as /other directories are referred to by name, and their names are separated by slashes (/). If a path refers to a directory it can end in / usually an extra slash at the end of a path makes no difference

Examples of Absolute Paths

NOTES

An absolute path starts at the root of the directory hierarchy, and names directories under it:
/etc/hostname
Meaning the file called hostname in the directory etc in the root directory.
We can use ls to list files in a specific directory by specifying the absolute path:
$ ls /usr/share/doc/

Current Directory

Your shell has a current directory — the directory in which you are currently working Commands like ls use the current directory if none is specified. Use the pwd (print working directory) command to see what your current directory is:
$ pwd /home/fred

Change the current directory with cd:

$ cd /mnt/cdrom
$ pwd /mnt/cdrom
Use cd without specifying a path to get back to your home directory

Making and Deleting Directories

The mkdir command makes new, empty, directories. For example, to make a directory for storing company accounts:
$ mkdir Accounts
To delete an empty directory, use rmdir:
$ rmdir OldAccounts
Use rm with the -r (recursive) option to delete directories and all the files they contain:
$ rm -r OldAccounts
Be careful — rm can be a dangerous tool if misused

Relative Paths

Paths don't have to start from the root directory. A path which doesn't start with / is a relative path. It is relative to some other directory, usually the current directory. For example, the following sets of directory changes both end up in the same directory:
$ cd /usr/share/doc
$ cd /
$ cd usr
$ cd share/doc

Relative paths specify files inside directories in the same way as absolute ones

Special Dot Directories

Every directory contains two special filenames which help making relative paths:

The directory points to the parent directory. ls. will list the files in the parent directory

For example, if we start from /home/fred:

$ cd ..
$ pwd
/home
$ cd ..
$ pwd
/

The special directory. points to the directory it is in. So./foo is the same file as foo

Using Dot Directories in Paths

The special .. and. directories can be used in paths just like any other directory name:

$ cd ../other-dir/

Meaning "the directory other-dir in the parent directory of the current directory"

It is common to see .. used to 'go back' several directories from the current directory:

$ ls ../../../../far-away-directory/

The directory is most commonly used on its own, to mean "the current directory"

Hidden Files

The special. and.. directories don't show up when you do ls. They are hidden files. Simple rule: files whose names start with. are considered 'hidden'. Make ls display all files, even the hidden ones, by giving it the -a (all) option:

$ ls -a. .. .bashrc .profile report.doc

Hidden files are often used for configuration files'. Usually found in a user's home directory

You can still read hidden files — they just don't get listed by ls by default

Paths to Home Directories

The symbol ~ (tilde) is an abbreviation for your home directory. So for user 'fred', the following are equivalent:

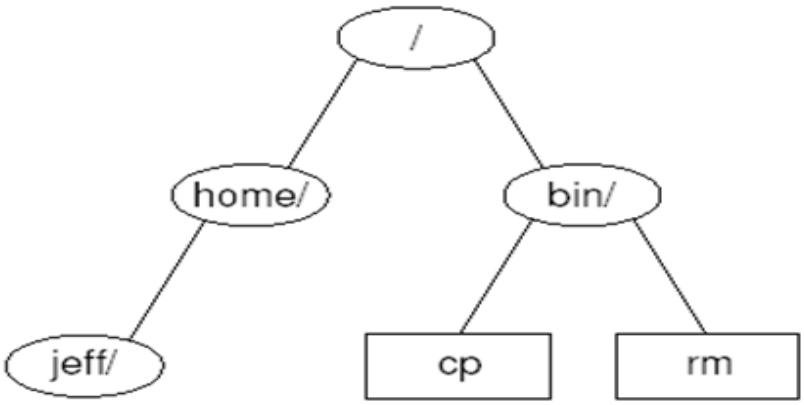
$ cd /home/fred/documents/
$ cd ~/documents/

The ~ is expanded by the shell, so programs only see the complete path. You can get the paths to other users' home directories using ~, for example:

$ cat ~alice/notes.txt
The following are all the same for user 'fred':
$ cd
$ cd ~
$ cd /home/fred

Basics of File

- A file is a place to store data: a possibly-empty sequence of bytes
- A directory is a collection of files and other directories
- Directories are organized in a hierarchy, with the root directory at the top. The root directory is referred to as /



File Extensions

- It's common to put an extension, beginning with a dot, on the end of a filename
- The extension can indicate the type of the file:
- .txt Text file
- .gif Graphics Interchange Format image
- .jpg Joint Photographic Experts Group image
- .mp3 MPEG-2 Layer 3 audio
- .gz Compressed file
- .tar Unix 'tape archive' file

.tar.gz, .tgz Compressed archive file

On UNIX, file extensions are just a convention. The kernel just treats them as a normal part of the name. A few programs use extensions to determine the type of a file

Filename Completion

- Modern shells help you type the names of files and directories by completing partial names
- Type the start of the name (enough to make it unambiguous) and press Tab

- For an ambiguous name (there are several possible completions), the shell can list the options:
- For Bash, type Tab twice in succession
- For C shells, type Ctrl+D
- Both of these shells will automatically escape spaces and special characters in the filenames

Copying Files with cp

- **Syntax**: cp [options] source-file destination-file
- Copy multiple files into a directory: cp files directory
- Common options:
- -f, force overwriting of destination files
- -i, interactively prompt before overwriting files
- -a, archive, copy the contents of directories recursively

Examples of cp

Copy /etc/smb.conf to the current directory:
$ cp /etc/smb.conf .
Create an identical copy of a directory called work, and call it work-backup:
$ cp -a work work-backup
Copy all the GIF and JPEG images in the current directory into images:
$ cp *.gif *.jpeg images/

Moving Files with mv

mv can rename files or directories, or move them to different directories
It is equivalent to copying and then deleting
But is usually much faster
Options:
-f, forces overwrite, even if target already exists
-i, ask user interactively before overwriting files
For example, to rename poetry.txt to poems.txt:
$ mv poetry.txt poems.txt
To move everything in the current directory somewhere else:
$ mv * ~/old-stuff/

Deleting Files with rm

rm deletes ('removes') the specified files
You must have written permission for the directory the file is in to remove it
Use carefully if you are logged in as root!
Options:
-f, delete write-protected files without prompting
-i, interactive — ask the user before deleting files
-r, recursively delete files and directories
For example, clean out everything in /tmp, without prompting to delete each file:

$ rm -rf /tmp/*

Identifying Types of Files

- The data in files comes in various different formats (executable programs, text files, etc.)
- The file command will try to identify the type of a file:
- $ file /bin/bash/bin/bash: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked (uses shared libs), stripped
- It also provides extra information about some types of file
- Useful to find out whether a program is actually a script:
- $ file /usr/bin/zless /usr/bin/zless: Bourne shell script text
- If file doesn't know about a specific format, it will guess:
- $ file /etc/passwd/etc/passwd: ASCII text

Finding Files with locate

- The locate command is a simple and fast way to find files
- For example, to find files relating to the email program mutt:
- $ locate mutt
- The locate command searches a database of filenames
- The database needs to be updated regularly
- Usually this is done automatically with cron
- But locate will not find files created since the last update
- The -i option makes the search case-insensitive
- -r treats the pattern as a regular expression, rather than a simple string

Directories and Renames

Programs under Linux are files, stored in directories like /bin and /usr/bin
Run them from the shell, simply by typing their name
Many programs take options, which are added after their name and prefixed with -
For example, the -l option to ls gives more information, including the size of files and the date they were last modified:
$ ls -l

drwxrwxr-x	2	fred	users	4096	Mar 01	10:57 Accounts
-rw-rw-r--	1	fred	users	345	Mar 01	10:57 notes.txt
-rw-r--r--	1	fred	users	3255	Mar 01	10:57 report.txt

Many programs accept filenames after the options
Specify multiple files by separating them with spaces
To copy the contents of a file into another file, use the cp command:
$ cp CV.pdf old-CV.pdf

To rename a file use the mv (‘move’) command:
$ mv commitee_minutes.txt committee_minutes.txt
Similar to using cp then rm
For both commands, the existing name is specified as the first argument and the new name as the second. If a file with the new name already exists, it is overwritten

NOTES

BLOCK 2

EX. NO:4 UNIX PERMISSIONS

Access to a file has three levels:

- Read permission – If authorized, the user can read the contents of the file.
- Write permission – If authorized, the user can modify the file.
- Execute permission – If authorized, the user can execute the file as a program.

Each file is associated with a set of identifiers that are used to determine who can access the file:

- User ID (UID) – Specifies the user that owns the file. By default, this is the creator of the file.
- Group ID (GID) – Specifies the user-group that the file belongs to.

Finally, there are three sets of access permissions associated with each file:

- User permission – Specifies the level of access given to the user matching the file's UID.
- Group permission – Specifies the level of access given to users in groups matching the file's GID.
- Others permission – Specifies the level of access given to users without a matching UID or GID.

Together, this scheme of access controls makes the Unix system extremely secure while simultaneously providing the flexibility required of a multi-user system.

The *ls -l* command can be used to view the permissions associated with each of the files in the current folder.

INODE

Each file in UNIX has a unique number called as an inode. Using this number the file information like user, group, ownership and access mode information can be found. A files inode number can be found using the following command:

Ls–i

If the inode number is known, the following command can be used to get details of the file:

Ls –l

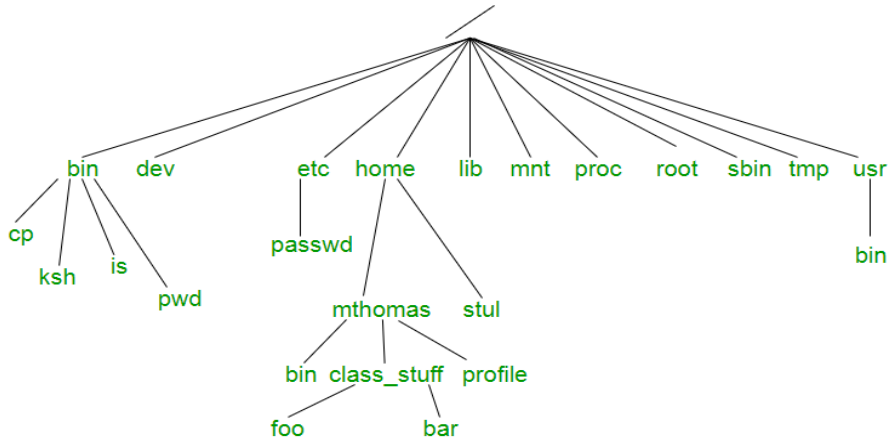
Directory Hierarchy

UNIX uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A UNIX filesystem is a collection of files and directories that has the following properties −

- It has a root directory (/) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an **inode**.
- By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
- It is self-contained. There are no dependencies between one filesystem and another.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix −



Sr.No.	Directory & Description
1	**/** This is the root directory which should contain only the directories needed at the top level of the file structure
2	**/bin** This is where the executable files are located. These files are available to all users
3	**/dev** These are device drivers

4	**/etc** Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages
5	**/lib** Contains shared library files and sometimes other kernel-related files
6	**/boot** Contains files for booting the system
7	**/home** Contains the home directory for users and other accounts
8	**/mnt** Used to mount other temporary file systems, such as **cdrom** and **floppy** for the **CD-ROM** drive and **floppy diskette drive**, respectively
9	**/proc** Contains all processes marked as a file by **process number** or other information that is dynamic to the system
10	**/tmp** Holds temporary files used between system boots
11	**/usr** Used for miscellaneous purposes, and can be used by many users. Includes administrative commands, shared files, library files, and others
12	**/var** Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
13	**/sbin** Contains binary (executable) files, usually for system administration. For example, ***fdisk*** and ***ifconfig*** utlities
14	**/kernel** Contains kernel files

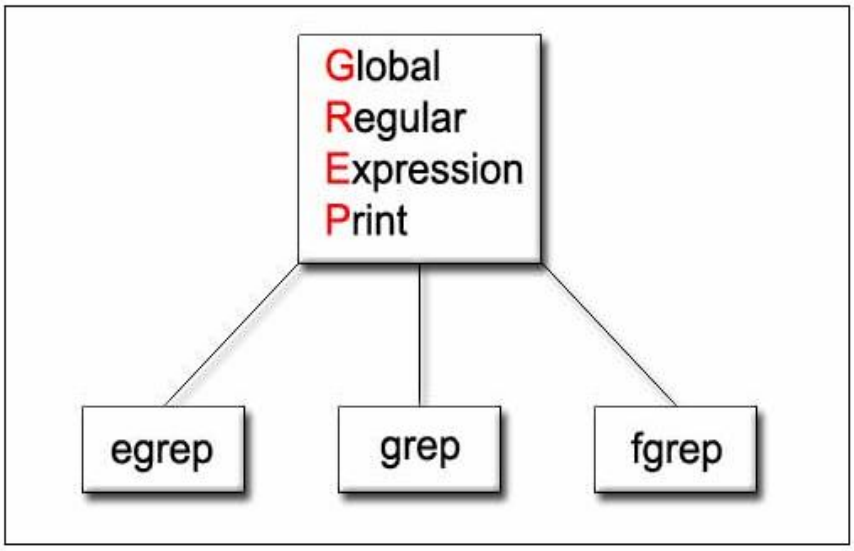
Device File

This is another special file that is used to describe a physical device, such as a printer or a portable drive. This file contains no data whatsoever; it merely maps any data coming its way to the physical device it describes. Device file types typically include: character device files, block device files, UNIX domain sockets, named pipes and symbolic links. However, not all of these file types may be present across various UNIX implementations.

Grep Family

In the simplest terms, grep (global regular expression print) is a small family of commands that search input files for a search string, and print the lines that match it. Although this may not seem like a terribly useful command at first, grep is considered one of the most useful commands in any UNIX system. Grep is made up of three separate, yet connected commands, grep, egrep, and fgrep, a sort of holy trinity of UNIX commands. All three of the grep commands work the same way. Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen. Grep will repeat this process until the file runs out of lines. Notice that nowhere in this process does grep store lines, change lines, or search only a part of a line.

The simplest possible example of grep is simply:



grep "boot" a file

In this example, grep would loop through every line of the file "a file" and print out every line that contains the text "boot." Since grep is named the "global regular expression print" it's not surprising that grep can also search for regular expressions in addition to normal strings. Regular expressions are searched for in the same way a normal string is. In fact, the strings we entered before were just very simple regular expressions.

grep "e$" a file

While grep supports a handful of regular expression commands, it does not support certain useful sequences such as the + and ? operators. If you would like to use these, you will have to use extended grep (egrep). Egrep is equivalent to grep -E, but as it is fairly common to want the extended functionality, egrep is also its own separate command.

egrep "boot|boots"

Fgrep is the third member of the grep family. It stands for "fast grep" and for good reason. Fgrep is faster than other grep commands because it does not interpret regular expressions, it only searches for strings of literal characters. Fgrep is equivalent to grep -F. If one fgreped for boot|boots, rather than interpreting that as a search for either the word boot or the word boots, frep would simply search for the literal string "boot|boots" in the file. For instance, with normal grep the following command would search for lines ending with the word "broken"

fgrep "broken$" a_file

However, we can see that with fgrep, it will return the line "broken$tuff" because it is not interpreting the dollar sign, only the entire string as literal characters.

Example
Now that we have skimmed over the basic functions of the commands in the grep family, we can look at a few examples of more advanced functionality. The following example is an example of grepping through the output of another program rather than a file. This particular example will print out the files that find returns that contain the text "hello
find | grep "hello"

Normally, grep does not have a way to search through portions of files, but when the file is first processed by another program, this is possible. This example performs a grep on the last 8 lines of a file
tail -n8 a file | grep "boo"

By using the exec switch with the find command, we can find files that contain the search string. The following will search for the string "boo" in every directory below the current directory

find . -exec grep "boo" {} \;
grep is the only command of the three that supports back references and saving. The following uses back references to find lines that contain two of the same lowercase letter in succession.

grep "\([a-z]\)\1" a file

Other Filters

A filter is a program that takes input from the standard input file, process and sends its output to the standard output file. The filters can be used in an efficient way. The following are some of the filters.
SORT: This command is used to sort the data's in some order.
Syntax: $sort<filename>
n
-r
Reverses the order of sort. Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs.
-f
Sorts upper and lowercase together.
+x
Ignores first x fields when sorting
HEAD: It is used to display the top ten lines of file.
Syntax: $head<filename>
TAIL: This command is used to display the last ten lines of file.
Syntax: $tail<filename>
PAGE: This command shows the page by page a screen full of information is displayed after
which the page command displays a prompt and passes for the user to strike the enter key to
continue scrolling.
Syntax: $ls –a\p
MORE: It also displays the file page by page. To continue scrolling with more command ,
press the space bar key.
Syntax: $more<filename>
GREP: This command is used to search and print the specified patterns from the file.
Syntax: $grep [option] pattern <filename>

The pg and more Commands

A long output can normally be zipped by you on the screen, but if you run text through more or use the pg command as a filter; the display stops once the screen is full of text. Let's assume that you have a long directory listing

Stream Editor

`sed` is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as `ed`), `sed` works by making only one pass over the input(s), and is consequently more efficient. But it is `sed`'s ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

`sed` commands

The following commands are supported in GNU sed. Some are standard POSIX commands, while other are GNU extensions. Details and examples for each command are in the following sections. (Mnemonics) are shown in parentheses.

a\
text

Append *text* after a line.

a *text*

Append *text* after a line (alternative syntax).

b *label*

Branch unconditionally to *label*. The *label* may be omitted, in which case the next cycle is started.

c\
text

Replace (change) lines with *text*.

c *text*

Replace (change) lines with *text* (alternative syntax).

d

Delete the pattern space; immediately start next cycle.

D

If pattern space contains newlines, delete text in the pattern space up to the first newline, and restart cycle with the resultant pattern space, without reading a new line of input.
If pattern space contains no newline, start a normal new cycle as if the d command was issued.

e

Executes the command that is found in pattern space and replaces the pattern space with the output; a trailing newline is suppressed.

e *command*

Executes *command* and sends its output to the output stream. The command can run across multiple lines, all but the last ending with a back-slash.

F

(filename) Print the file name of the current input file (with a trailing newline).

g

Replace the contents of the pattern space with the contents of the hold space.

G

Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.

h

(hold) Replace the contents of the hold space with the contents of the pattern space.

H

Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.

i

text

insert *text* before a line.

EX. NO: 5 AWK PATTERN SCANNING AND PROCESSING LANGUAGE

Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

Program Structure

The basic operation of awk is to scan a set of input lines one after another, searching for lines that match any of a set of patterns or conditions that the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern. Accordingly, an awk program is a sequence of pattern-action statements of the form
pattern { action }
pattern { action }
...
The third program in the abstract,
$1 == "address" {print $2, $3 } is a typical example, consisting of one pattern-action statement. Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action (which may involve multiple steps) is executed. Then the next line is read and the matching starts over. This process typically continues until all the input has been read. Either the pattern or the action in a pattern-action statement may be omitted. If there is no action with a pattern, as in
$1 == "name" the matching line is printed. If there is no pattern with an action, as in {print $1, $2} then the action is performed for every input line. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Usage

There are two ways to run an awk program. You can type the command awk ’pattern-action statements’ optional list of input files to execute the

pattern-action statements on the set of named input files. For example, you could say awk ' { print $1, $2 }' data1 data2

If no files are mentioned on the command line, the awk interpreter will read the standard input. Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line. The arrangement above is convenient when the awk program is short (a few lines). If the program is long, it is often more convenient to put it into a separate file, say my program, and use the -f option to fetch it awk -f my program optional list of input files. Any filename can be used in place of my program

Fields

Awk normally reads its input one line at a time; it splits each line into a sequence of fields, where, by default, a field is a string of non-blank, non-tab characters. As input for many of the awk programs in this manual, we will use the following file, countries. Each line contains the name of a country, its area in thousands of square miles, its population in millions, and the continent where it is, for the ten largest countries in the world. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.)

USSR 8650 262 Asia
Canada 3852 24 North America
China 3692 866 Asia
USA 3 615 219 North America
Brazil 3286 116 South America
Australia 2968 14 Australia
India 1269 637 Asia
Argentina 1072 26 South America
Sudan 968 19 Africa
Algeria 920 18 Africa

The wide space between fields is a tab in the original input; a single blank separates North and South from America. This file is typical of the kind of data that awk is good at processing a mixture of words and numbers separated into fields by blanks and tabs. The number of fields in a line is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, in which case the first line of countries would have 4 fields, the second 5, and so on. It's possible to set the field separator to just tab, so each line would have 4 fields, matching the meaning of the data; we'll show how to do this shortly. For the time being, we'll use the default: fields separated by blanks and/or tabs.

The first field within a line is called $1, the second $2, and so forth. The entire line is called $0.

Printing

If the pattern in a pattern-action statement is missing, the action is executed for all input lines. The simplest action is to print each line; this can be accomplished by the awk program consisting of a single print statement:
{ print } (P.1)
so the command
awk '{ print }' countries
prints each line of countries, thus copying the file to the standard output. In the remainder of this paper, we shall only show awk programs, without the command line that invokes them. Each complete program is identified by (P.n) in the right margin; in each case, the program can be run either by enclosing it in quotes as the first argument of the awk command as shown above, or by putting it in a file and invoking awk with the -f flag, as discussed in Section 1.2. In an example, if no input is mentioned, it is assumed to be the file countries.
The print statement can be used to print parts of a record; for instance, the program { print $1, $3 } prints the first and third fields of each input line. Thus awk' { print $1, $3 }' countries.

Built-In Variables

Besides reading the input and splitting it into fields, awk counts the number of lines read and the number of fields within the current line; you can use these counts in your awk programs. The variable NR is the number of the current input line, and NF is the number of fields. So, the program {print NR, NF} prints the number of each line and how many fields it has, while {print NR, $0} prints each line preceded by its line number.

Simple Patterns

You can select specific lines for printing or other processing with simple patterns. For example, the operators == tests for equality. To print the lines for which the fourth field equals the string Asia we can use the program consisting of the single pattern:
$4 == "Asia" (P.6)
With the file countries as input, this program yields
USSR 8650 262 Asia
China 3692 866 Asia
India 1269 637 Asia
The complete set of comparisons is >, >=, <, <=, == (equal to) and!= (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose we want to print only countries with more than 100 million populations. The program
$3 > 100 is all that is needed (remember that the third field is the population in millions); it prints all lines in which the third field exceeds 100.You can also use patterns called ''regular expressions'' to select lines. The simplest form of a regular expression is a string of characters enclosed in slashes:
/US/

This program prints each line that contains the (adjacent) letters US anywhere; with file countries as input, it prints

USSR 8650 262 Asia
USA 3615 219 North America

We will have a lot more to say about regular expressions in §2.4.

There are two special patterns, BEGIN and END, that ‘‘match’’ before the first input line has been read and after the last input line has been processed. This program uses BEGIN to print a title:

BEGIN {print "Countries of Asia:"}
/Asia/ {print " ", $1} The output is
Countries of Asia:
USSR
China
India

Patterns

In a pattern-action statement, the pattern is an expression that selects the input lines for which the associated action is to be executed. This section describes the kinds of expressions that may be used as patterns.

BEGIN and END

The special pattern BEGIN matches before the first input record is read, so any statements in the action part of a BEGIN are done once before awk starts to read its first input file. The pattern END matches the end of the input, after the last file has been processed. BEGIN and END provide a way to gain control for initialization and wrap up.

The field separator is stored in a built-in variable called FS. Although FS can be reset at any time, usually the only sensible place is in a BEGIN section, before any input has been read. For example, the following awk program uses BEGIN to set the field separator to tab (\t) and to put column headings on the output. The second printf statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings.

The END action prints the totals.

BEGIN {FS = "\t"
printf "%10s %6s %5s %s\n",
"COUNTRY", "AREA", "POP", "CONTINENT"}
{printf "%10s %6d %5d %s\n", $1, $2, $3, $4
area = area + $2; pop = pop + $3}
END {printf "\n%10s %6d %5d\n", "TOTAL", area, pop} (P.20)

With the file countries as input, produces

COUNTRY AREA POP CONTINENT

USSR	8650	262	Asia
Canada	3852	24	North America

China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

TOTAL 30292 2201

Combinations of Patterns

A compound pattern combines simpler patterns with parentheses and the logical operator's || (or), && (and)! (Not). For example, suppose we wish to print all countries in Asia with a population of more than 500 million. The following program does this by selecting all lines in which the fourth field is Asia and the third field exceeds 500:

$4 == "Asia" && $3 > 500 (P.32)
The program
$4 == "Asia" || $4 == "Africa" (P.33)

selects lines with Asia or Africa as the fourth field. Another way to write the latter query is to use a regular expression with the alternation operator |:
$4 ~ /^(Asia|Africa)$/ (P.34)
The negation operator! has the highest precedence, then &&, and finally ||. The operators && and || evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

A pattern range consists of two patterns separated by a comma, as in pat 1, pat 2 {...} In this case, the action is performed for each line between an occurrence of pat 1 and the next occurrence of pat 2 (inclusive). As an example, the pattern
- 10 -
/Canada/, /Brazil/ (P.35)
Matches lines starting with the first line that contains Canada up through the next occurrence of Brazil:
Canada 3852 24 North America
China 3692 866 Asia
USA 3615 219 North America
Brazil 3286 116 South America
Similarly, since FNR is the number of the current record in the current input file, the program FNR == 1, FNR == 5 {print FILENAME, $0 } prints the first five records of each input file with the name of the current input file prepended.

1. **AWK Operations:**
   (a) Scans a file line by line
   (b) Splits each input line into fields
   (c) Compares input line/fields to pattern
   (d) Performs action(s) on matched lines

2. **Useful For:**
   (a) Transform data files
   (b) Produce formatted reports

3. **Programming Constructs:**
   (a) Format output lines
   (b) Arithmetic and string operations
   (c) Conditionals and loops

Files and good Filters

Unix provides a number of powerful commands to process texts in different ways. These text processing commands are often implemented as filters. Filters are commands that always read their input from 'stdin' and write their output to 'stdout'. Users can use file redirection and 'pipes' to setup 'stdin' and 'stdout' as per their need. Pipes are used to direct the 'stdout' stream of one command to the 'stdin' stream of the next command. Some standard filter commands are described below. These commands may also take an input file as a parameter, but by default when the file is not specified, they operate as filter command.

Working with Text Files

Unix-like systems are designed to manipulate text very well. The same techniques can be used with plain text, or text-based formats Most Unix configuration files are plain text

Text is usually in the ASCII character set Non-English text might use the ISO-8859 character sets Unicode is better, but unfortunately many Linux command-line utilities don't (directly) support it yet. Lines of Text files are naturally divided into lines. In Linux a line ends in a line feed character 10, hexadecimal 0x0A, Other operating systems use different combinations

Windows and DOS use a carriage return followed by a line feed Macintosh systems use only a carriage return Programs are available to convert between the various formats.

Filtering Text and Piping

The UNIX philosophy: use small programs, and link them together as needed

Each tool should be good at one specific job

Join programs together with pipes

Indicated with the pipe character: |

The first program prints text to its standard output

That gets fed into the second program's standard input

For example, to connect the output of echo to the input of wc:

$ echo "count these words, boy" | wc

Displaying Files with less

If a file is too long to fit in the terminal, display it with less:
$ less README
less also makes it easy to clear the terminal of other things, so is useful even for small files. Often used on the end of a pipe line, especially when it is not known how long the output will be:
$ wc *.txt | less
Doesn't choke on strange characters, so it won't mess up your terminal (unlike cat). Counting Words and Lines with wc
wc counts characters, words and lines in a file
If used with multiple files, outputs counts for each file, and a combined total

Options:

-c output character count
-l output line count
-w output word count
Default is -clw
Examples: display word count for essay.txt:
$ wc -w essay.txt
Display the total number of lines in several text files:
$ wc -l *.txt

Sorting Lines of Text with sort

The sort filter reads lines of text and prints them sorted into order
For example, to sort a list of words into dictionary order:
$ sort words > sorted-words
The -f option makes the sorting case-insensitive
The -n option sorts numerically, rather than lexicographically

Removing Duplicate Lines with unique

Use unique to find unique lines in a file
Removes consecutive duplicate lines
Usually give it sorted input, to remove all duplicates
Example: find out how many unique words are in a dictionary:
$ sort /usr/dict/words | uniq | wc -w
sort has a -u option to do this, without using a separate program:
$ sort -u /usr/dict/words | wc -w
sort | uniq can do more than sort -u, though:
uniq -c counts how many times each line appeared
uniq -u prints only unique lines
uniq -d prints only duplicated lines

Selecting Parts of Lines with cut

Used to select columns or fields from each line of input

Select a range of
Characters, with -c
Fields, with -f
Field separator specified with -d (defaults to tab)
A range is written as start and end position: e.g., 3-5
Either can be omitted
The first character or field is numbered 1, not 0
Example: select usernames of logged in users:
$ who | cut -d" " -f1 | sort -u

Expanding Tabs to Spaces with expand

Used to replace tabs with spaces in files
Tab size (maximum number of spaces for each tab) can be set with -t number
Default tab size is 8
To only change tabs at the beginning of lines, use -i
Example: change all tabs in foo.txt to three spaces, display it to the screen:
$ expand -t 3 foo.txt
$ expand -3 foo.txt

Using fmt to Format Text Files

Arranges words nicely into lines of consistent length
Use -u to convert to uniform spacing
One space between words, two between sentences
Use -w width to set the maximum line width in characters
Defaults to 75
Example: change the line length of notes.txt to a maximum of 70 characters, and display it on the screen:
$ fmt -w 70 notes.txt | less

Reading the Start of a File with head

Prints the top of its input, and discards the rest
Set the number of lines to print with -n lines or -lines
Defaults to ten lines
View the headers of a HTML document called homepage.html:
$ head homepage.html
Print the first line of a text file (two alternatives):
$ head -n 1 notes.txt
$ head -1 notes.txt

Reading the End of a File with tail

Similar to head, but prints lines at the end of a file
The -f option watches the file forever
Continually updates the display as new entries are appended to the end of the file
Kill it with Ctrl+C
The option -n is the same as in head (number of lines to print)

Example: monitor HTTP requests on a webserver:
$ tail -f /var/log/httpd/access_log

Numbering Lines of a File with nl or cat

Display the input with line numbers against each line
There are options to finely control the formatting
By default, blank lines aren't numbered
The option -ba numbers every line
cat -n also numbers lines, including blank ones

Dumping Bytes of Binary Data with od

Prints the numeric values of the bytes in a file
Useful for studying files with non-text characters
By default, prints two-byte words in octal
Specify an alternative with the -t option
Give a letter to indicate base: o for octal, x for hexadecimal, u for unsigned decimal, etc.
Can be followed by the number of bytes per word
Add z to show ASCII equivalents alongside the numbers
A useful format is given by od -t x1z — hexadecimal, one byte words, with ASCII
Alternatives to od include xxd and hexdump

Paginating Text Files with pr

Convert a text file into paginated text, with headers and page fills
Rarely useful for modern printers

Options:
-d double spaced output
-h header change from the default header to header
-l lines change the default lines on a page from 66 to lines
-o width set ('offset') the left margin to width
Example:
$ pr -h "My Thesis" thesis.txt | lpr

Dividing Files into Chunks with split

Splits files into equal-sized segments
Syntax: split [options] [input] [output-prefix]
Use -l n to split a file into n-line chunks
Use -b n to split into chunks of n bytes each
Output files are named using the specified output name with aa, ab, ac, etc., added to the end of the prefix
Example: Split essay.txt into 30-line files, and save the output to files short_aa, short_ab, etc:
$ split -l 30 essay.txt short_

Using split to Span Disks

If a file is too big to fit on a single floppy, Zip or CD-ROM disk, it can be split into small enough chunks
Use the -b option, and with the k and m suffixes to give the chunk size in kilobytes or megabytes
For example, to split the file database.tar.gz into pieces small enough to fit on Zip disks:
$ split -b 90m database.tar.gz zip-
Use cat to put the pieces back together:
$ cat zip-* > database.tar.gz

Reversing Files with tac

Similar to cat, but in reverse
Prints the last line of the input first, the penultimate line second, and so on
Example: show a list of logins and logouts, but with the most recent events at the end:
$ last | tac

Translating Sets of Characters with tr

tr translates one set of characters to another
Usage: tr start-set end-set
Replaces all characters in start-set with the corresponding characters in end-set
Cannot accept a file as an argument, but uses the standard input and output
Options:
-d deletes characters in start-set instead of translating them
-s replaces sequences of identical characters with just one (squeezes them)
tr Examples
Replace all uppercase characters in input-file with lowercase characters (two alternatives):
$ cat input-file | tr A-Z a-z
$ tr A-Z a-z < input-file
Delete all occurrences of z in story.txt:
$ cat story.txt | tr -d z
Run together each sequence of repeated f characters in lullaby.txt to with just one f:
$ tr -s f < lullaby.txt

EX NO: 6 WILDCARD CHARACTERS

Wildcards (also referred to as meta characters) are symbols or special characters that represent other characters. You can use them with any command such as ls command or rm command to list or remove files matching a given criteria, receptively.

These wildcards are interpreted by the shell and the results are returned to the command you run. There are three main wildcards in UNIX:

- An asterisk **(*)** – matches one or more occurrences of any character, including no character.
- Question mark **(?)** – represents or matches a single occurrence of any character.
- Bracketed characters **([])** – matches any occurrence of character enclosed in the square brackets. It is possible to use different types of characters (alphanumeric characters): numbers, letters, other special characters etc.

BLOCK 3

EX. NO: 7 BASIC UNIX COMMANDS

File and Directory Related commands

pwd

This command prints the current working directory

ls

This command displays the list of files in the current working directory.

$ls –l Lists the files in the long format
$ls –t Lists in the order of last modification time
$ls –d Lists directory instead of contents
$ls -u Lists in order of last access time

cd

This command is used to change from the working directory to any other directory specified.

$cd directoryname

cd..

This command is used to come out of the current working directory.

$cd ..

mkdir

This command helps us to make a directory.

$mkdir directoryname

rmdir

This command is used to remove a directory specified in the command line. It requires the specified directory to be empty before removing it.

$rmdir directoryname

cat

This command helps us to list the contents of a file we specify.

$cat [option][file]
cat > filename – This is used to create a new file.
cat >>filename – This is used to append the contents of the file

cp

This command helps us to create duplicate copies of ordinary files.

$cp source destination

mv

This command is used to move files.

$mv source destination

ln

This command is to establish an additional filename for the same ordinary file.

$ln firstname secondname

rm

This command is used to delete one or more files from the directory.

$rm [option] filename

$rm –i Asks the user if he wants to delete the file mentioned.

$rm –r Recursively delete the entire contents of the directory as well as the directory itself.

Process and status information commands

who

This command gives the details of who all have logged in to the UNIX system currently.

$ who

who am i

This command tells us as to when we had logged in and the system's name for the connection being used.

$who am i

date

This command displays the current date in different formats.

+%D	mm/dd/yy	+%w	Day of the week
+%H	Hr-00 to 23	+%a	Abbr.Weekday
+%M	Min-00 to 59	+%h	Abbr.Month
+%S	Sec-00 to 59	+%r	Time in AM/PM
+%T	HH:MM:SS	+%y	Last two digits of the year

echo

This command will display the text typed from the keyboard.

$echo

Eg **:** $echo Have a nice day

O/p **:** Have a nice day

Text related commands

head

This command displays the initial part of the file. By default it displays first ten lines of the file.

$head [-count] [filename]

tail

This command displays the later part of the file. By default it displays last ten lines of the file.

$tail [-count] [filename]

wc

This command is used to count the number of lines, words or characters in a file.

$wc [-lwc] filename

find

The find command is used to locate files in a directory and in a subdirectory.

The –name option
This lists out the specific files in all directories beginning from the named directory. Wild cards can be used.

The –type option
This option is used to identify whether the name of files specified are ordinary files or directory files. If the name is a directory then use "-type d " and if it is a file then use “-type f”.

The –mtime option
This option will allow us to find that file which has been modified before or after a specified time. The various options available are –mtime n(on a particular day),-mtime +n(before a particular day),-mtime –n(after a particular day)

The –exec option
This option is used to execute some commands on the files that are found by the find command.

File Permission commands

chmod

Changes the file/directory permission mode: $ chmod 777 file1
Gives full permission to owner, group and others
$ chmod o-w file1
Removes write permission for others.

Useful Commands:

exit - Ends your work on the UNIX system.

EX. NO: 8 HISTORY OF UNIX SHELLS

The independence of the shell from the UNIX operating system *per se* has led to the development of dozens of shells throughout UNIX history—although only a few have achieved widespread use.

The first major shell was the Bourne shell (named after its inventor, Steven Bourne); it was included in the first popular version of UNIX, Version 7, starting in 1979. The Bourne shell is known on the system as *sh*. Although UNIX has gone through many, many changes, the Bourne shell is still popular and essentially unchanged. Several UNIX utilities and administration features depend on it.

The first widely used alternative shell was the C shell, or *csh*. This was written by Bill Joy at the University of California at Berkeley as part of the Berkeley Software Distribution (BSD) version of UNIX that came out a couple of years after Version 7. It's included in most recent UNIX versions.

The C shell gets its name from the resemblance of its commands to statements in the C Programming Language, which makes the shell easier for programmers on UNIX systems to learn. It supports a number of operating system features (e.g., job control) that were unique to BSD UNIX but by now have migrated to most other modern versions. It also has a few important features (e.g., aliases) that make it easier to use in general.

In recent years a number of other shells have become popular. The most notable of these is the Korn shell. This shell is a commercial product that incorporates the best features of the Bourne and C shells, plus many features of its own. The Korn shell is similar to *bash* in most respects; both have an abundance of features that make them easy to work with. The advantage of*bash* is that it is free

THE BOURNE AGAIN SHELL

The Bourne Again shell (named in punning tribute to Steve Bourne's shell) was created for use in the GNU project. The GNU project was started by Richard Stallman of the Free Software Foundation (FSF) for the purpose of creating a UNIX-compatible operating system and replacing all of the commercial UNIX utilities with freely distributable ones. GNU embodies not only new software utilities, but a new distribution concept: the copy left. Copulated software may be freely distributed so long as no restrictions are placed on further distribution (for example, the source code must be made freely available).

Bash, intended to be the standard shell for the GNU system, was officially "born" on Sunday, January 10, 1988. Brian Fox wrote the original versions of bash and read line and continued to improve the shell up until 1993. Early in 1989 he was joined by Chet Ramey, who was responsible for numerous bug fixes and the inclusion of many useful features. Chet Ramey is now the official maintainer of bash and continues to make further enhancements.

In keeping with the GNU principles, all versions of bash since 0.99 have been freely available from the FSF. Bash has found its way onto every major version of UNIX and is rapidly becoming the most popular Bourne shell derivative. It is the standard shell included with Linux, a widely used free UNIX operating system.

In 1995 Chet Ramey began working on a major new release, 2.0, which was released to the public for the first time on December 23, 1996. Bash 2.0 adds a range of new features to the old release and brings the shell into better compliance with various standards.

This book describes the latest release of bash 2.0 (version 2.01, dated June 1997). It is applicable to all previous releases of bash. Any features of the current release that are different in, or missing from, previous releases will be noted in the text.

FEATURES OF BASH

Although the Bourne shell is still known as the "standard" shell, bash is becoming increasingly popular. In addition to its Bourne shell compatibility, it includes the best features of the C and Korn shells as well as several advantages of its own.

Bash's command-line editing modes are the features that tend to attract people to it first. With command-line editing, it's much easier to go back and fix mistakes or modify previous commands than it is with the C shell's history mechanism—and the Bourne shell doesn't let you do this at all.

Ex. No: 9 DECIDING ON A SHELL

A shell is a program that provides the traditional, text-only user interface for Unix-like operating systems. Its primary function is to read commands that are typed into a console (i.e., an all-text display mode) or terminal window (an all-text window) in a GUI (graphical user interface) and then execute (i.e., run) them.

Although bash, the default shell on many Debian based Linux distros like Ubuntu and Linux Mint, is highly versatile and can be used for almost anything, each shell has its own characteristics and there might be situations in which it is preferable to use some other shell, such as ash, csh, ksh, sh or zsh. For example, the csh shell has a syntax that resembles that of the highly popular C programming language, and thus it is sometimes preferred by programmers

How to find out what shell you are running

type at prompt:

```
$ echo $SHELL
```

Here is what you may see:

/bin/sh : This is the Bourne shell.

/bin/ksh93 : This is the Korn shell.

/bin/bash : This is the Bash shell.

/bin/zsh : This is the Z shell.

/bin/csh : This is the C Shell.

/bin/tcsh : This is the TC Shell.

Valid login Shells

$ vi /etc/shells

Change your shell

The UNIX shell is most people's main access to the UNIX operating system and as such any improvement to it can result in considerably more effective use of the system, and may even allow you to do things you couldn't do before. The primary improvement most of the new generation shells give you is increased speed. They require fewer key strokes to get the same results due to their completion features, they give you more information (e.g. showing your directory in your prompt, showing which

files it would complete) and they cover some of the more annoying features of UNIX, such as not going back up symbolic links to directories.

How to Change Shells

There are two different methods of changing shells.

1. Temporarily changing your shell for the session you are in.
2. Changing your login shell which is permanent.

Temporarily changing your shell for the session you are in

Use csh as again an example. I will assume you already have csh installed on your system. All you need to do is type the command below into your terminal.

```
$ csh
```

There are a few ways to verify this. **This first comes from above.**

```
$ ps
```

If you have csh installed on your system you will see something similar to this.

```
PID    TTY          TIME CMD
13964  pts/0        00:00:00 bash
22450  pts/0        00:00:00 csh
22451   pts/0       00:00:00 ps
```

Upon changing shells, a different command prompt (i.e., the short text message at the left of each command line) may be shown, depending on the previous and new shells. For example, if the original shell were bash and the new shell is sh, the command prompt would change for a user George from something like [george@localhost george]$ to something like sh-2.05b$.

To return to the original shell, or any other one for that matter, all that is necessary is to just type its name and then press the ENTER key. Thus, for example, to return to the bash shell from the sh shell (or from any other shell), all that is required is to type the word **bash** or **exit**.

Changing your login shell which is permanent

Use a program called **chsh**. There is a interactive method and non-interactive method. Type this into your terminal.

INTERACTIVE METHOD

```
$ chsh
```

This results in a brief dialog in which the user is prompted first for its password and then for the full path of the desired new shell.

In particular, it is important to first test the shell temporarily in the current session and then to make certain that a valid shell name is being entered when making the permanent change.

NON-INTERACTIVE METHOD

Use csh as again an example.

$ chsh -s /bin/csh

The -s sets it for you without having to go into the editor to do it.

Block 4

Ex. No: 10 SHELL COMMAND FILES

A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one. A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a command prompt (usually $), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

- A Shell is a program whose primary purpose is to read commands and run other programs.
- The Shell's main advantages are its high action –to-keystroke ratio, its support for automating repetitive tasks and its capacity to access networked machines.
- The Shell's main disadvantage are its primarily textual nature and how cryptic its commands and operation can be.

Navigation and file Directory

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which forms a directory tree
- cd path changes the current working directory
- Is path prints a listing of a specific file or directory; ls on its own lists the current working directory
- pwd prints the user's current working directory.
- / on its own is the root directory of the whole file system.
- A relative path specifies a location starting from the current location
- An absolute path specifies a location from the root of the file system.
- Directory names in a path are separated with / on Unix, but \ on Windows.
- .. means 'the directory above the current one'; . on its own means 'the current directory'

Working with file and directory

• cp old new copies a file.
• mkdir path creates a new directory.
•mv old new moves (renames) a file or directory.
•rm path removes (deletes) a file.
•* matches zero or more characters in a filename, so *.txt matches all files ending in .txt.
•? matches any single character in a filename, so ?.txt matches a.txt but not any.txt.
•Use of the Control key may be described in many ways, including Ctrl-X, Control-X, and ^X.
•The shell does not have a trash bin: once something is deleted, it's really gone.
•Most files' names are something extension. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
•Depending on the type of work you do, you may need a more powerful text editor than Nano.

Loops

• A for loop repeats commands once for everything in a list.
• Every for loop needs a variable to refer to the thing it is currently operating on.
• Use $name to expand a variable (i.e., get its value). ${name} can also be used.
• Do not use spaces, quotes, or wildcard characters such as '*' or '?' in filenames, as it complicates variable expansion.
• Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
• Use the up-arrow key to scroll up through previous commands to edit and repeat them.
• Use Ctrl-R to search through the previously entered commands.
• Use history to display recent commands, and ! number to repeat a command by number.

Bourne Shell

The prompt for this shell is $ and its derivatives are listed below:

- POSIX shell also is known as sh
- Korn Shell also knew as sh
- Bourne Again SHell also knew as bash (most popular)

EX. NO: 11 BOURNE SHELL PROGRAMMING

The shbang line
The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a `#!` followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

Example

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

Comments

Comments are descriptive material preceded by a `#` sign. They are in effect until the end of a line and can be started anywhere on the line.

Example

```
# this text is not
# interpreted by the shell
```

Wildcards

There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

EXAMPLE

Filename expansion:

rm *; ls ??; cat file[1-3];

Quotes protect metacharacter:

echo "How are you?"

Displaying output

To print output to the screen, the echo command is used. Wildcards must be escaped with either a backslash or matching quotes.
EXAMPLE
echo "What is your name?"

Local variables

Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

EXAMPLE

variable_name=value
name="John Doe"
x=5

Global variables

Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

EXAMPLE

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Extracting values from variables

To extract the value from variables, a dollar sign is used.

EXAMPLE

```
echo $variable_name
echo $name
echo $PATH
```

Reading user input

The `read` command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The `read` command can accept muliple variable names. Each variable will be assigned a word.

E*XAMPLE*

```
echo "What is your name?"
read name
read name1 name2 ...
```

Arguments (positional parameters)

Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.

EXAMPLE

At the command line:

```
$ scriptname arg1 arg2 arg3 ...
```

In a script:

```
echo $1 $2 $3     Positional parameters
echo $*             All the positional paramters
echo $#           The number of positional parameters
```

Arrays (positional parameters)

The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in set command, and the words are accessed by position. Up to nine positions are allowed. The built-in shift command shifts off the first word on the left-hand side of the list. The individual words are accessed by position values starting at 1.

EXAMPLE

```
set word1 word2 word3
echo $1 $2 $3                Displays word1, word2, and word3
set apples peaches plums
shift                              Shifts off apples
echo $1                          Displays first element of the list
echo $2                          Displays second element of the list
echo $*                          Displays all elements of the list
```

Command substitution

To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, backquotes are used.

EXAMPLE

```
variable_name=`command`
echo $variable_name
now=`date`
echo $now
echo "Today is `date`"
```

Arithmetic

The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations.

EXAMPLE

```
n=`expr 5 + 5`
echo $n
```

Operators

The Bourne shell uses the built-in test command operators to test numbers and strings.

EXAMPLE

Equality:

= string
!= string
-eq number
-ne number

Logical:

-a and
-o or
! not

Relational:

-gt greater than
-ge greater than, equal to
-lt less than
-le less than, equal to

Loops

There are three types of loops: while, until and for.The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

E*XAMPLE*

```
while command
do
  block of statements
done

while [ expression ]
do
  block of statements
done
until command              for variable in word1 word2 word3 ...
do                      do
  block of statements      block of statements
done                    done
     until [ expression ]
do
   block of statements
done
```

The Bourne Shell Script

```
1  #!/bin/sh
2  # The Party Program—Invitations to friends from the "guest" file
3  guestfile=/home/jody/ellie/shell/guests
4  if [ ! –f "$guestfile" ]
5  then
6       echo "`basename $guestfile` non–existent"
7       exit 1
8  fi
9  PLACE="Sarotini's"; export PLACE
10 Time=`date +%H`
Time=`expr $Time + 1`
11 set cheese crackers shrimp drinks "hot dogs" sandwiches
12 for person in `cat $guestfile`
do
13      if [ $person =~ root ]
then
14          continue
15      else
16      #     mail –v –s "Party" $person <<- FINIS
17          cat <<-FINIS
Hi ${person}! Please join me at $PLACE for a party!
Meet me at $Time o'clock.
I'll bring the ice cream. Would you please bring $1 and
anything else you would like to eat? Let me know if you
can make it. Hope to see you soon.
Your pal,
ellie@`hostname`
FINIS
18          shift
19          if [ $# –eq 0 ]
then
20            set cheese crackers shrimp drinks "hot dogs" sandwiches
fi
fi
21  done
echo "Bye..."
```

EXPLANATION

This line lets the kernel know that you are running a Bourne shell script.

This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

The variable guestfile is set to the full pathname of a file called guests.

This line reads: If the file guests does not exist, then print to the screen "guests nonexistent" and exit from the script.

The then is usually on a line by itself, or on the same line as the if statement if it is preceded by a semicolon.

The UNIX basename command removes all but the filename in a search path. Because the command is enclosed in backquotes, command substitution will be performed and the output displayed by the echo command.

If the file does not exist, the program will exit. An exit with a value of 1 indicates that there was a failure in the program.

The fi keyword marks the end of the block of if statements.

Variables are assigned the values for the place and time. PLACE is an environment variable, because after it is set, it is exported.

The value in the Time variable is the result of command substitution; i.e., the output of the date +%H command (the current hour) will be assigned to Time.

The list of foods to bring is assigned to special variables (positional parameters) with the set command.

The for loop is entered. It loops through until each person listed in the guest file has been processed.

If the variable person matches the name of the user root, loop control will go to the top of the for loop and process the next person on the list. The user root will not get an invitation.

The continue statement causes loop control to start at line 12, rather than continuing to line 16.The blocks of statements under else are executed if line 13 is not true.

The mail message is sent when this line is uncommented. It is a good idea to comment this line until the program has been thoroughly debugged, otherwise the e-mail will be sent to the same people every time the script is tested.

The next statement, using the cat command with the here document, allows the script to be tested by sending output to the screen that would normally be sent through the mail when line 7 is uncommented. After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, ensuring that each person is assigned a food.

The value of $# is the number of positional parameters left. If that number is 0, the food list is empty. The food list is reset. The done keyword marks the end of the block of statements in the body of the for loop

EX. NO: 12 SHELLS PROGRAMMING ON FILES

Basic File Attributes - Read, Write and Execute

There are three basic attributes for plain file permissions: read, write, and execute.

Read Permission of a file

If you have read permission of a file, you can see the contents. That means you can
use more, cat, etc.

Write Permission of a file

If you have to write permission of a file, you can change the file. This means you can
Add to a file, or overwrite a file. You can empty a file called "yourfile" by copying
The empty (/dev/null) file on top of it
Cat /dev/null yourfile

Execute Permission of a file

If the file has execute permission, then you can ask the operating system to run the file as if it were a program. If it's a binary file/program, you can execute it like any other program.
If the file is a shell script, then the execute attribute says you can treat it as if it were a program. To put it another way, you can create a file using your favorite editor, add the execute attribute to it, and it "becomes" a program. However, since a shell has to read the file, a shell script has to be readable and executable. A compiled program does not need to be readable.

The basic permission characters, "r", "w", and "x"

r means read w means write, and x means execute.

Using chmod to change permissions

The chmod command is used to change permission. The simplest way to use the chmod command is to add or subtract the permission to a file. A simple plus or minus is used to add or subtract the permission.
You may want to prevent yourself from changing an important file. Remove the write permission of the file "myfile" with the command chmod -w myfile If you want to make file "myscript" executable, type chmod +x myscript You can add or remove more than one of these attributes at a time chmod -rwx file
chmod +wx file

You can also use the "=" to set the permission to an exact combination. This command removes the write and executes permission, while adding the read permission:

chmod =r myfile

Note that you can change permissions of files you own. That is, you can remove all permissions of a file, and then add them back again. You can make a file "read only" to protect it. However, making a file read only does not prevent you from deleting the file. That's because the file is in a directory, and directories also have read, write and execute permission. And the rules are different.

Basic Directory Attributes - Read, Write and Search

Directories use these same permissions, but they have a different meaning. Yes, very different meanings.

Read permission on a directory

If a directory has read permission, you can see what files are in the directory. That is, you can do an "ls" command and see the files inside the directory. However, read permission of a directory does not mean you can read the contents of files in the directory.

Write permission on a directory

Write permission means you can add a new file to the directory. It also means you can rename or move files in the directory.

Execute permission on a directory

Execute allows you to use the directory name when accessing files inside that directory. The "x" permission means the directory is "searchable" when searching for executables. If it's a program, you can execute the program.

File Test Operators

There are following operators to test various properties associated with a Unix file. Assume a variable file holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

Example:

```
#!/bin/sh
file="/home /jnec /test.sh"
if [ -r $file ]
then
 echo "File has read access"
else
 echo "File does not have read access"
fi
if [ -w $file ]
then
 echo "File has write permission"
else
 echo "File does not have write permission"
fi
if [ -x $file ]
then
```

```
 echo "File has execute permission"
else
 echo "File does not have execute permission"
fi
if [ -f $file ]
then
15
 echo "File is an ordinary file"
else
 echo "This is sepcial file"
fi
if [ -d $file ]
then
 echo "File is a directory"
else
 echo "This is not a directory"
fi
if [ -s $file ]
then
 echo "File size is zero"
else
 echo "File size is not zero"
fi
if [ -e $file ]
then
 echo "File exists"
else
 echo "File does not exist"
fi
```

NOTES

BLOCK 5

EX. NO 13 MENU DRIVEN FILE HANDLING

There is more than one method to create a menu within a shell script. Using select offers an easy way to create menus and modify them as needed. Before we use select, to show you an alternative method of implementing a menu in a shell script using an infinite while loop, case statement and a bunch of echo and read statements.

Problem Statement
Write a menu driven program in C to add, display, search, update and delete the student record

Program

```
#include<stdio.h>
      #include<conio.h>
      #include<process.h>
      #include<stdio.h>
      #include<string.h>
      #include<process.h>
      #include<conio.h>

      void main( )
      {
       FILE  *fp, *ft;
       char  another,choice;
       clrscr();

       struct Student
       {

        int roll;
        char  name[40];
        int  age;
       };

       struct Student  stu;
       char  Studentname[40] ;
       char found='n';
       long int  recsize ;
```

```
    int roll;

    fp = fopen ( "E:\\Student.DAT", "rb+" ) ;

    if ( fp == NULL )
    {
     fp = fopen ( "E:\\Student.DAT", "wb+" ) ;

     if ( fp == NULL )
     {
       puts ( "Cannot open file" ) ;
       exit(0) ;
     }
    }

    recsize = sizeof (stu) ;

    while ( 1 )
    {
     clrscr() ;

   gotoxy ( 30, 6 ) ;
   printf ( "1. Add Records" ) ;
   gotoxy ( 30, 8 ) ;
   printf ( "2. List Records" ) ;
   gotoxy ( 30, 10) ;
   printf ( "3. Search Records" ) ;
   gotoxy ( 30, 12 ) ;
   printf ( "4. Modify Records" ) ;
   gotoxy ( 30, 14 ) ;
   printf ( "5. Delete Records" ) ;
   gotoxy ( 30, 16 ) ;
   printf ( "6. Exit" ) ;
   gotoxy ( 30, 18 ) ;
   printf ( "Your choice : " ) ;

fflush (stdin) ;

choice = getche();
switch( choice )
{
        case '1' :

         fseek ( fp, 0 , SEEK_END ) ;
         another = 'Y' ;

         while ( another == 'Y' || another == 'y')
         {
```

```
            clrscr();
            printf ( "\n\tEnter Roll : " ) ;
            scanf ( "%d",&stu.roll);

            printf ( "\n\tEnter Name : " ) ;
            scanf ( "%s",&stu.name);

            printf ( "\n\tEnter Age : " ) ;
            scanf ( "%d",&stu.age);

            fwrite ( &stu, recsize, 1, fp ) ;

            printf ( "\n\tAdd another Record (Y/N) " ) ;
            fflush ( stdin ) ;
            another = getche( ) ;
           }

           break ;

          case '2' :

           rewind ( fp ) ;
            clrscr();
            printf ( "\n\tRoll\tName\t\tAge");
           while ( fread ( &stu, recsize, 1, fp ) == 1 )
            printf ( "\n\t%d\t%s\t\t%d", stu.roll, stu.name, stu.age );

            printf("\n\n\n\tPRESS ANY KEY TO EXIT.....");
            getch();

           break ;


          case '3' :

           rewind ( fp ) ;
           clrscr();
           printf("\nEnter Roll for Search : ");
           scanf("%d",&roll);

           while ( fread ( &stu, recsize, 1, fp ) == 1 )
           {

                        if(roll == stu.roll)
                        {
                          printf ( "\n\tRoll\tName\t\tAge");
                          printf ( "\n\t%d\t%s\t\t%d", stu.roll, stu.name,
stu.age );
```

```
                found='y';
                break;
                }
 }

 if(found=='n')
 printf("\n\n\n\tNo Match Found.....");

 printf("\n\n\n\tPRESS ANY KEY TO EXIT.......");
 getch();

break ;



case '4' :


 rewind ( fp ) ;

clrscr();
printf("\nEnter Roll for Modify : ");
scanf("%d",&roll);

 while ( fread ( &stu, recsize, 1, fp ) == 1 )
 {


        if ( roll == stu.roll )
        {
                printf ( "\n\tEnter New Roll : " ) ;
                scanf ( "%d",&stu.roll);

                printf ( "\n\tEnter New Name : " ) ;
                scanf ( "%s",&stu.name);

                printf ( "\n\tEnter New Age : " ) ;
                scanf ( "%d",&stu.age);


         fseek ( fp, - recsize, SEEK_CUR ) ;

         fwrite ( &stu, recsize, 1, fp ) ;
          printf ( "\nData Updated :  " ) ;
         break ;
```

```
                }
            }


            printf("\n\n\n\tPRESS ANY KEY TO EXIT.....");
            getch();

            break ;

           case '5' :

            another = 'Y' ;

            clrscr();
            printf ( "\nEnter Roll to Delete :  " ) ;
            scanf ( "%d",&roll ) ;

            ft = fopen ( "E:\\temp.DAT", "w" ) ;

            rewind ( fp ) ;
            while ( fread ( &stu, recsize, 1, fp ) == 1 )
            {
                    if ( roll != stu.roll )
                     fwrite ( &stu, recsize, 1, ft ) ;
            }

            fclose ( fp ) ;
            fclose ( ft ) ;
            remove ( "E:\\Student.DAT" ) ;
            rename ( "E:\\temp.DAT", "E:\\Student.DAT" ) ;

            fp = fopen ( "E:\\Student.DAT", "rb+" ) ;

            printf("\n\n\n\tPRESS ANY KEY TO EXIT.....");
            getch();


            break ;

           case '6' :
            fclose ( fp ) ;
            exit(0) ;
  }
 }
}
```

Ex.No14 Menu Driven Shell Programming

NOTES

Write a script to make Following File and Directory Management Operations menu based
a] Display Current dirctory
b] List Directory
c] Make Directory
d] Change Directory
e] Copy A File
f] Rename A File
G] Delete A File
H] Edit a file

```
echo  "\t\t\tMenu\n
   1]  Display Current dirctory\n
   2]  List Directory\n
   3]  Make Directory\n
   4]  Change Directory\n
   5]  Copy A File\n
   6]  Rename A File\n
   7] Delete A File\n
   8] Edit a file\n
   Enter Your Choice:- \c"
   read choice

case"$choice"in
   1) pwd ;;
   2) ls ;;
   3) echo "Enter the Name Of Directory :-\c"
      read dir
      if [-z $dir]; then
         echo "you have not enter the dir"elseif [ ! -e $dir]; then
       mkdir $dir
      else
       echo "Dir Already Exists"if;;
   4) echo "Enter the Name Of Directory :-\c"
      read dir
      if [-z $dir]; then
         echo "you have not enter the dir"elseif [ ! -e $dir ];then
        echo "Dir Not Found"else
       cd $dir
      if;;
   5) echo "Enter Soures File:-\c"
      read sfile
```

```
    if [-z $sfile];then
       Echo "Enter the Destination file"
       read dfile
       if [-z $dfile];then
           cp -i $sfile $dfiile
       else
           echo "You have not Enter Destination"ifelse
     echo "You have not entered Sorce File"if ;;
  6)echo "Enter Soures File:-\c"
    read sfile
    if [-z $sfile];then
       Echo "Enter the Destination file"
       read dfile
       if [-z $dfile];then
           mv -i $sfile $dfiile
       else
           echo "You have not Enter Destination"ifelse
     echo "You have not entered Sorce File"if ;;
  7)echo "Enter the File"
    read delfile
    if [-z $delfile] ; then
       echo "you have not Entered File"elseif [ ! -e $delfile]
         echo "File Does not Exist"else
      echo "Confirm Delete [y/n] :- \c"
       read  delans
        if [$delans -e y -o delans -e Y];then
          rm $delfile
       ififi;;
  8)exit
esac
```

UNIX SHELL PROGRAMMING

MODEL LAB EXCERCISES

UNIX SHELL PROGRAMMING

MODEL LAB EXCERCISES

SUM OF N GIVEN NUMBERS

AIM

To write a shell script to find the sum of n numbers.

ALGORITHM

Step 1: Start the program

Step 2: Enter the value of n

Step 3: Declare the variables of i and sum and both are initialized to zero.

Step 4: Perform the addition of n numbers using while loop

Step 5: Print the sum.

Step 6 : Stop the program

Program

```
echo "Sum of n natural numbers \n” echo "Enter the value of n "
read n i = 0
sum = 0 while[$i –lt $n] do
sum="expr $sum + expr $i" i = „expr $i + 1"
done
echo "Sum of n natural numbers are : $sum”
```

OUTPUT

$ sh sum.sh

```
Sum of n natural numbers
Enter the value of n : 3
Sum of n natural numbers are : 6
```

Result:

Thus, the shell script to find the sum of n natural numbers is entered and its output was verified.

SWAP TWO GIVEN NUMBERS

AIM

To write Unix shell program to swap two given numbers

ALGORITHM

1. Start the program.
2. Read the variables a, b.
3. Interchange the values of a and b using another temporary variable c as follows:

   c = a a = b b = c
4. Print the a and b.
5. Stop the program.

Program

```
echo “swapping using temporary variable” echo “enter a”
read  a
echo “enter b” read b
c=$a a=$b b=$c
echo “after swapping” echo “$a”
echo “$b”
```

OUTPUT

```
Enter a 10
Enter b  20
After swapping 20 10
```

Result:

Thus, the shell script to swap two numbers is entered and its output was verified.

LEAP YEAR CHECKING

AIM

To write a shell program to check whether the given year is leap year or not.

ALGORITHM

1. Start the program.
2. Read the year.
3. Check whether year%4,year%100,year%400 is zero.
4. If zero then print year is leap year.
5. Else print year is not leap year.
6. Stop the program.

Program

```
echo "Finding Leap Year" echo "enter any year" read y
a=`expr $y % 4` b=`expr $y % 100` c=`expr $y % 400`
if [ $a -eq 0 -a $b -ne 0 -o $c -eq 0 ] then
echo "$y is a leap year " else
echo "$y is not a leap year " fi
```

OUTPUT

```
Enter a year:
2000
Year is leap year
```

Result:

Thus, the shell script to check whether the given year is leap year or not is entered and its output was verified.

FINDING FACTORIAL

AIM

To Write shell program to find the factorial of given N value.

ALGORITHM

1. Start the program.
2. Read the number as n.
3. For every iteration until n<1 compute f=f*n.
4. Print the factorial of the given number as f.
5. Stop the program.

Program

```
echo “enter a positive number” read n
f=1
until [ $n -lt 1  ] do
f=`expr $f \* $n` n=`expr $n - 1` done
echo “factorial is $f”
```

OUTPUT

```
Enter positive number 4
Factorial is 24
```

Result:

Thus, the shell script to find the factorial of a given number is entered and its output was verified.

SUM OF DIGITS OF A GIVEN NUMBER

AIM:

To write a shell program for finding the sum of digits of a given number.

Sum = 1 + 2 + 3 + 4 +….+ N

ALGORITHM

1. Start the program.
2. Read the number as n.
3. Initialise sum=0
4. For every iteration n> 0

   compute rem=n%10

   n=n/10 sum=sum + rem
5. Print the sum of digits of the given number as sum.
6. Stop the program.

Program

```
echo “enter the number” read n
sum=0
while [ $n -gt  0  ] do
rem=`expr $n % 10` n=`expr $n / 10` sum=`expr $sum + $rem` done
echo “sum of digits is:$sum”
```

OUTPUT

Enter a number:1234 Sum of digits is:10

Result:

Thus, the shell script to find the sum the digits of a given number is entered and its output was verified.

GREATEST AMONG THREE NUMBERS

AIM:

To write a shell program for finding the greatest among three numbers.

ALGORITHM

1. Start the program.
2. Read the three numbers a, b, c.
3. Check whether a is greater than b and c.
4. If yes then print a is big.
5. Else check whether b is greater than c.
6. If yes then print b is big.
7. Else print c is big.
8. Stop the program.

Program

```
echo “enter a b c” read a
read b read  c
if [ $a -gt $b ] && [  $a  -gt  $c  ] then
echo “$a big”
elif [ $b -gt $c ] then
echo “$b big” else
echo “$c big”
```

OUTPUT

Enter a b c 10 20 30

C big

Result:

Thus, the shell script to find the sum the digits of a given number is entered and its output was verified.

B.C.A.
10154
UNIX & SHELL PROGRAMMING LAB

V - Semester

ALAGAPPA UNIVERSITY

[Accredited with A+' Grade by NAAC (CGPA:3.64) in the Third Cycle and Graded as Category–I University by MHRD-UGC]

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION