Unix Tutorial

UNIX TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Unix Tutorial

UNIX is a computer Operating System which is capable of handling activities from multiple users at the same time.

Unix was originated around in 1969 at AT&T Bell Labs by Ken Thompson and Dennis Ritchie. This tutorial gives a very good understanding on Unix.

Audience

This tutorial has been prepared for the beginners to help them understand them basic to advanced concepts covering Unix commands, UNIX shell scripting and various utilities.

Prerequisites

We assume you have little knowledge about Operating System and its functionalities. A basic understanding on various computer concepts will also help you in understanding various exercises given in this tutorial.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

Unix Getting Started

The UNIX operating system is capable of handling activities from multiple users at the same time.

What is Unix ?

The UNIX operating system is a set of programs that act as a link between the computer and the user.

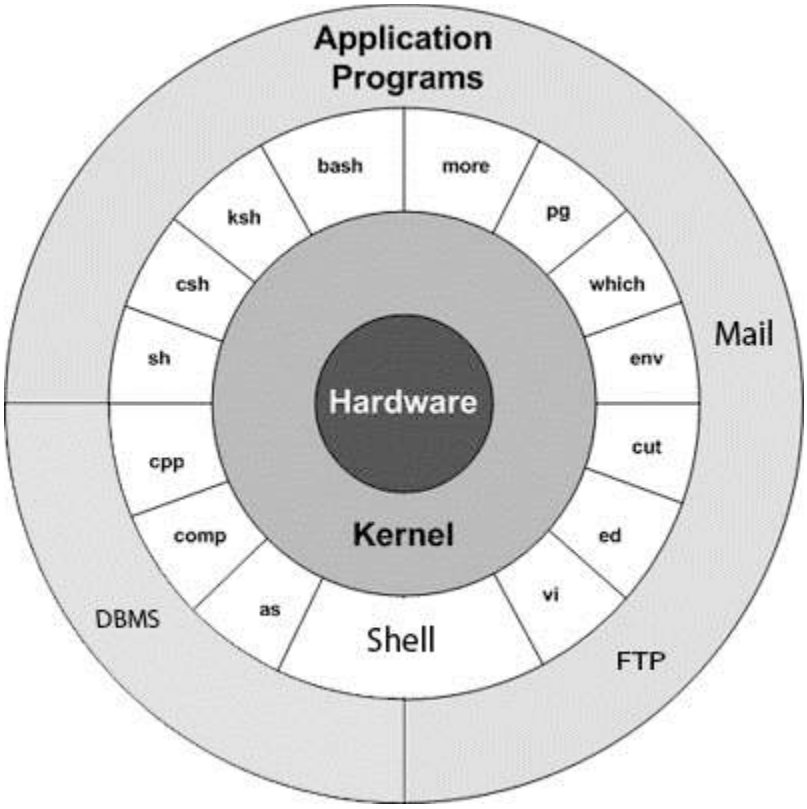
The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel.

Users communicate with the kernel through a program known as the shell. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence UNIX is called multitasking.

Unix Architecture:

Here is a basic block diagram of a UNIX system:



The main concept that unites all versions of UNIX is the following four basics:

- **Kernel:** The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, tash scheduling and file management.

- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.

- **Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.

- **Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

System Bootup:

If you have a computer which has UNIX operating system installed on it, then you simply need to turn on its power to make it live.

As soon as you turn on the power, system starts booting up and finally it prompts you to log into the system, which is an activity to log into the system and use it for your day to day activities.

Login Unix:

When you first connect to a UNIX system, you usually see a prompt such as the following:

```
login:
```

To log in:

1. Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.

2. Type your userid at the login prompt, then press ENTER. Your userid is case-sensitive, so be sure you type it exactly as your system administrator instructed.

3. Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.

4. If you provided correct userid and password then you would be allowed to enter into the system. Read the information and messages that come up on the screen something as below.

```
login : amrood
amrood's password:
Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73
$
```

You would be provided with a command prompt (sometime called $ prompt) where you would type your all the commands. For example to check calendar you need to type **cal** command as follows:

```
$ cal
     June 2009
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30

$
```

Change Password:

All Unix systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from hackers and crackers. Here are the steps to change your password:

1. To start, type **passwd** at command prompt as shown below.

2. Enter your old password the one you're currently using.

3. Type in your new password. Always keep your password complex enough so that no body can guess it. But make sure, you remember it.

4. You would need to verify the password by typing it again.

```
$ passwd
Changing password for amrood
(current) Unix password:******
```

```
New UNIX password:*******
Retype new UNIX password:*******
passwd: all authentication tokens updated  successfully

$
```

Note: I have put stars (*) just to show you the location where you would need to enter the current and new passwords otherwise at your system, it would not show you any character when you would type.

Listing Directories and Files:

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

You can use **ls** command to list out all the files or directories available in a directory. Following is the example of using **ls** command with **-l** option.

```
$ ls -l
total 19621
drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov 25  2007 usr
-rwxr-xr-x  1 root   root        3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf

$
```

Here enteries starting with **d.....** represent directories. For example uml, univ and urlspedia are directories and rest of the enteries are files.

Who Are You?

While you're logged in to the system, you might be willing to know : **Who am I**?

The easiest way to find out "who you are" is to enter the **whoami** command:

```
$ whoami
 amrood

$
```

Try it on your system. This command lists the account name associated with the current login. You can try **who am i** command as well to get information about yourself.

Who is Logged In?

Sometime you might be interested to know who is logged in to the computer at the same time.

There are three commands are available to get you this information, based on how much you'd like to learn about the other users: **users, who,** and **w**.

```
$ users
 amrood bablu qadir
```

```
$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu  ttyp2 Oct 4 09:08 (calliope)
qadir  ttyp4 Oct 8 12:09 (dent)

$
```

Try **w** command on your system to check the output. This would list down few more information associated with the users logged in the system.

Logging Out:

When you finish your session, you need to log out of the system to ensure that nobody else accesses your files while masquerading as you.

To log out:

1. Just type **logout** command at command prompt, and the system will clean up everything and break the connection

System Shutdown:

The most consistent way to shut down a Unix system properly via the command line is to use one of the following commands:

Command	Description
halt	Brings the system down immediately.
init 0	Powers off the system using predefined scripts to synchronize and clean up the system prior to shutdown
init 6	Reboots the system by shutting it down completely and then bringing it completely back up
poweroff	Shuts down the system by powering off.
reboot	Reboots the system.
shutdown	Shuts down the system.

You typically need to be the superuser or root (the most privileged account on a Unix system) to shut down the system, but on some standalone or personally owned Unix boxes, an administrative user and sometimes regular users can do so.

CHAPTER

2

Unix File Management

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

When you work with UNIX, one way or another you spend most of your time working with files. This tutorial would teach you how to create and remove files, copy and rename them, create links to them etc.

In UNIX there are three basic types of files:

1. **Ordinary Files:** An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.

2. **Directories:** Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders.

3. **Special Files:** Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

Listing Files:

To list the files and directories stored in the current directory. Use the following command:

```
$ls
```

Here is the sample output of the above command:

```
$ls

bin         hosts   lib      res.03
ch07        hw1     pub      test_results
ch07.bak    hw2     res.01   users
docs        hw3     res.02   work
```

The command **ls** supports the **-1** option which would help you to get more information about the listed files:

```
$ls -l
total 1962188

drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
```

```
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov 25  2007 usr
drwxr-xr-x  2    200    300      4096 Nov 25  2007 webthumb-1.01
-rwxr-xr-x  1 root   root        3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
$
```

Here is the information about all the listed columns:

1. First Column: represents file type and permission given on the file. Below is the description of all type of files.

2. Second Column: represents the number of memory blocks taken by the file or directory.

3. Third Column: represents owner of the file. This is the Unix user who created this file.

4. Fourth Column: represents group of the owner. Every Unix user would have an associated group.

5. Fifth Column: represents file size in bytes.

6. Sixth Column: represents date and time when this file was created or modified last time.

7. Seventh Column: represents file or directory name.

In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

Prefix	Description
-	Regular file, such as an ASCII text file, binary executable, or hard link.
b	Block special file. Block input/output device file such as a physical hard drive.
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories.
l	Symbolic link file. Links on any regular file.
p	Named pipe. A mechanism for interprocess communications
s	Socket used for interprocess communication.

Meta Characters:

Meta characters have special meaning in Unix. For example ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark **?** matches with single character.

For Example:

```
$ls ch*.doc
```

Displays all the files whose name start with ch and ends with .doc:

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here * works as meta character which matches with any character. If you want to display all the files ending with just .doc then you can use following command:

```
$ls *.doc
```

Hidden Files:

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files:

- **.profile:** the Bourne shell (sh) initialization script
- **.kshrc:** the Korn shell (ksh) initialization script
- **.cshrc:** the C shell (csh) initialization script
- **.rhosts:** the remote shell configuration file

To list invisible files, specify the -a option to ls:

```
$ ls -a

.          .profile       docs     lib     test_results
..         .rhosts        hosts    pub     users
.emacs     bin            hw1      res.01  work
.exrc      ch07           hw2      res.02
.kshrc     ch07.bak       hw3      res.03
$
```

- Single dot **.**: This represents current directory.
- Double dot **..**: This represents parent directory.

Note: I have put stars (*) just to show you the location where you would need to enter the current and new passwords otherwise at your system, it would not show you any character when you would type.

Creating Files:

You can use **vi** editor to create ordinary files on any Unix system. You simply need to give following command:

```
$ vi filename
```

Above command would open a file with the given filename. You would need to press key **i** to come into edit mode. Once you are in edit mode you can start writing your content in the file as below:

```
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
```

Once you are done, do the following steps:

- Press key **esc** to come out of edit mode.
- Press two keys **Shift + ZZ** together to come out of the file completely.

Now you would have a file created with **filemame** in the current directory.

```
$ vi filename
$
```

Editing Files:

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows:

```
$ vi filename
```

Once file is opened, you can come in edit mode by pressing key **i** and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key **esc** and then you can use following keys to move inside a file:

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move up side in the file.
- **j** key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use **i** key to come in edit mode. Edit the file, once you are done press **esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

Display Content of a File:

You can use **cat** command to see the content of a file. Following is the simple example to see the content of above created file:

```
$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
$
```

You can display line numbers by using **-b** option along with **cat** command as follows:

```
$ cat filename -b
1   This is unix file....I created it for the first time.....
2   I'm going to save this content in this file.
$
```

Counting Words in a File:

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is the simple example to see the information about above created file:

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns:

1. First Column: represents total number of lines in the file.
2. Second Column: represents total number of words in the file.
3. Third Column: represents total number of bytes in the file. This is actual size of the file.

4. Fourth Column: represents file name.

You can give multiple files at a time to get the information about those file. Here is simple syntax:

```
$ wc filename1 filename2 filename3
```

Copying Files:

To make a copy of a file use the **cp** command. The basic syntax of the command is:

```
$ cp source_file destination_file
```

Following is the example to create a copy of existing file **filename**.

```
$ cp filename copyfile
$
```

Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

Renaming Files:

To change the name of a file use the **mv** command. Its basic syntax is:

```
$ mv old_file new_file
```

Following is the example which would rename existing file **filename** to **newfile**:

```
$ mv filename newfile
$
```

The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory.

Deleting Files:

To delete an existing file use the **rm** command. Its basic syntax is:

```
$ rm filename
```

Caution: It may be dangerous to delete a file because it may contain useful information. So be careful while using this command. It is recommended to use **-i** option along with **rm** command.

Following is the example which would completely remove existing file **filename**:

```
$ rm filename
$
```

You can remove multiple files at a tile as follows:

```
$ rm filename1 filename2 filename3
$
```

Standard Unix Streams:

Under normal circumstances every Unix program has three streams (files) opened for it when it starts up:

1. **stdin :** This is referred to as *standard input* and associated file descriptor is 0. This is also represented as STDIN. Unix program would read default input from STDIN.

2. **stdout :** This is referred to as *standard output* and associated file descriptor is 1. This is also represented as STDOUT. Unix program would write default output at STDOUT

3. **stderr :** This is referred to as *standard error* and associated file descriptor is 2. This is also represented as STDERR. Unix program would write all the error message at STDERR.

CHAPTER

3

Unix Directories

A directory is a file whose sole job is to store file names and related information. All files whether ordinary, special, or directory, are contained in directories.

UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character (/), and all other directories are contained below it.

Home Directory:

The directory in which you find yourself when you first login is called your home directory.

You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command:

```
$cd ~
$
```

Here **~** indicates home directory. If you want to go in any other user's home directory then use the following command:

```
$cd ~username
$
```

To go in your last directory you can use following command:

```
$cd -
$
```

Absolute/Relative Pathnames:

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /.

These are some example of absolute filenames.

```
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/Os3
```

A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user amrood' home directory, some pathnames might look like this:

```
chem/notes
personal/res
```

To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory:

```
$pwd
/user0/home/amrood

$
```

Listing Directories:

To list the files in a directory you can use the following syntax:

```
$ls dirname
```

Following is the example to list all the files contained in /usr/local directory:

```
$ls /usr/local

X11       bin          gimp       jikes       sbin
ace       doc          include    lib         share
atalk     etc          info       man         ami
```

Creating Directories:

Directories are created by the following command:

```
$mkdir dirname
```

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command:

```
$mkdir mydir
$
```

Creates the directory mydir in the current directory. Here is another example:

```
$mkdir /tmp/test-dir
$
```

This command creates the directory test-dir in the /tmp directory. The **mkdir** command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, mkdir creates each of the directories. For example:

```
$mkdir docs pub
$
```

Creates the directories docs and pub under the current directory.

Creating Parent Directories:

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, mkdir issues an error message as follows:

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test";
No such file or directory
$
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example:

```
$mkdir -p /tmp/amrood/test
$
```

Above command creates all the required parent directories.

Removing Directories:

Directories can be deleted using the **rmdir** command as follows:

```
$rmdir dirname
$
```

Note: To remove a directory make sure it is empty which means there should not be any file or sub-directory inside this directory.

You can create multiple directories at a time as follows:

```
$rmdir dirname1 dirname2 dirname3
$
```

Above command removes the directories dirname1, dirname2, and dirname2 if they are empty. The rmdir command produces no output if it is successful.

Changing Directories:

You can use the **cd** command to do more than change to a home directory: You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as follows:

```
$cd dirname
$
```

Here, dirname is the name of the directory that you want to change to. For example, the command:

```
$cd /usr/local/bin
$
```

Changes to the directory /usr/local/bin. From this directory you can cd to the directory /usr/home/amrood using the following relative path:

```
$cd ../../home/amrood
$
```

Renaming Directories:

The mv (move) command can also be used to rename a directory. The syntax is as follows:

```
$mv olddir newdir
$
```

You can rename a directory **mydir** to **yourdir** as follows:

```
$mv mydir yourdir
$
```

The directories . (dot) and .. (dot dot)

The filename . (dot) represents the current working directory; and the filename .. (dot dot) represent the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories files and use the -a option to list all the files and the -l option provides the long listing, this is the result.

```
$ls -la
drwxrwxr-x    4    teacher   class   2048  Jul 16 17.56 .
drwxr-xr-x    60   root              1536  Jul 13 14:18 ..
----------    1    teacher   class   4210  May 1 08:27 .profile
-rwxr-xr-x    1    teacher   class   1948  May 12 13:42 memo
$
```

CHAPTER

4

Unix File Permission Setup

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

- **Owner permissions:** The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions:** The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions:** The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators:

While using **ls -l** command it displays various information related to file permission as follows:

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here first column represents different access mode ie. permission associated with a file or directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x):

- The first three characters (2-4) represent the permissions for the file's owner. For example **-rwxr-xr--** represents that onwer has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example **-rwxr-xr--** represents that group has read (r) and execute (x) permission but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example **-rwxr-xr--** represents that other world has read (r) only permission.

File Access Modes:

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which are described below:

1. Read:

Grants the capability to read ie. view the contents of the file.

2. Write:

Grants the capability to modify, or remove the content of the file.

3. Execute:

User with execute permissions can run a file as a program.

Directory Access Modes:

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

1. Read:

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

2. Write:

Access means that the user can add or delete files to the contents of the directory.

3. Execute:

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.

A user must have execute access to the **bin** directory in order to execute ls or cd command.

Changing Permissions:

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

Using chmod in Symbolic Mode:

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
```

```
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g=r-x testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you could combine these commands on a single line:

```
$chmod o+wx,u-x,g=r-x testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions:

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups:

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files:

1. **chown:** The chown command stands for "change owner" and is used to change the owner of a file.
2. **chgrp:** The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership:

The chown command changes the ownership of a file. The basic syntax is as follows:

```
$ chown user filelist
```

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

Following example:

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

NOTE: The super user, root, has the unrestricted capability to change the ownership of a any file but normal users can change only the owner of files they own.

Changing Group Ownership:

The chrgp command changes the group ownership of a file. The basic syntax is as follows:

```
$ chgrp group filelist
```

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

Following example:

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group.

SUID and SGID File Permission:

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the **passwd** command, your new password is stored in the file /etc/shadow.
As a regular user, you do not have read or write access to this file for security reasons, but when you change your password, you need to have write permission to this file. This means that the **passwd** program has to give you additional permissions so that you can write to the file /etc/shadow.

Additional permissions are given to programs via a mechanism known as the Set User ID (SUID) and Set Group ID (SGID) bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is true for SGID as well. Normally programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter "s" if the permission is available. The SUID "s" bit will be located in the permission bits where the owners execute permission would normally reside. For example, the command

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1   root   bin  19031 Feb 7 13:47  /usr/bin/passwd*
$
```

Which shows that the SUID bit is set and that the command is owned by the root. A capital letter S in the execute position instead of a lowercase s indicates that the execute bit is not set.

If the sticky bit is enabled on the directory, files can only be removed if you are one of the following users:

- The owner of the sticky directory
- The owner of the file being removed
- The super user, root

To set the SUID and SGID bits for any directory try the following:

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root  4096 Jun 19 06:45 dirname
$
```

CHAPTER

5

Unix Environment

An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variables TEST and then we access its value using **echo** command:

```
$TEST="Unix Programming"
$echo $TEST
Unix Programming
```

Note that environment variables are set without using $ sign but while accessing them we use $sign as prefix. These variables retain their values until we come out shell.

When you login to the system, the shell undergoes a phase called initialization to set up various environments. This is usually a two step process that involves the shell reading the following files:

- /etc/profile
- profile

The process is as follows:

1. The shell checks to see whether the file **/etc/profile** exists.
2. If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
3. The shell checks to see whether the file **.profile** exists in your home directory. Your home directory is the directory that you start out in after you log in.
4. If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.

As soon as both of these files have been read, the shell displays a prompt:

```
$
```

This is the prompt where you can enter commands in order to have them execute.

Note - The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

The .profile File:

The file **/etc/profile** is maintained by the system administrator of your UNIX machine and contains shell initialization information required by all users on a system.
The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes

- The type of terminal you are using
- A list of directories in which to locate commands
- A list of variables effecting look and feel of your terminal.

You can check your **.profile** available in your home directory. Open it using **vi** editor and check all the variables set for your environment.

Setting the Terminal Type:

Usually the type of terminal you are using is automatically configured by either the **login** or **getty**programs. Sometimes, the autoconfiguration process guesses your terminal incorrectly.

If your terminal is set incorrectly, the output of commands might look strange, or you might not be able to interact with the shell properly.

To make sure that this is not the case, most users set their terminal to the lowest common denominator as follows:

```
$TERM=vt100
$
```

Setting the PATH:

When you type any command on command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually it is set as follows:

```
$PATH=/bin:/usr/bin
$
```

Here each of the individual entries separated by the colon character, :, are directories. If you request the shell to execute a command and it cannot find it in any of the directories given in the PATH variable, a message similar to the following appears:

```
$hello
hello: not found
$
```

There are variables like PS1 and PS2 which are discussed in the next section.

PS1 and PS2 Variables:

The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.

For example, if you issued the command:

```
$PS1='=>'
=>
=>
=>
```

Your prompt would become =>. To set the value of PS1 so that it shows the working directory, issue the command:

```
=>PS1="[\u@\h \w]\$"
[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$
[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$
```

The result of this command is that the prompt displays the user's username, the machine's name (hostname), and the working directory.

There are quite a few escape sequences that can be used as value arguments for PS1; try to limit yourself to the most critical so that the prompt does not overwhelm you with information.

Escape Sequence	Description
\t	Current time, expressed as HH:MM:SS.
\d	Current date, expressed as Weekday Month Date
\n	Newline.
\s	Current shell environment.
\W	Working directory.
\w	Full path of the working directory.
\u	Current user.s username.
\h	Hostname of the current machine.
\#	Command number of the current command. Increases with each new command entered.
\$	If the effective UID is 0 (that is, if you are logged in as root), end the prompt with the # character; otherwise, use the $.

You can make the change yourself every time you log in, or you can have the change made automatically in PS1 by adding it to your **.profile** file.

When you issue a command that is incomplete, the shell will display a secondary prompt and wait for you to complete the command and hit Enter again.

The default secondary prompt is > (the greater than sign), but can be changed by re-defining the **PS2**shell variable:

Following is the example which uses the default secondary prompt:

```
$ echo "this is a
> test"
this is a
test
$
```

Following is the example which re-define PS2 with a customized prompt:

```
$ PS2="secondary prompt->"
$ echo "this is a
secondary prompt->test"
this is a
test
$
```

Environment Variables:

Following is the partial list of important environment variables. These variables would be set and accessed as mentioned above:

Variable	Description
DISPLAY	Contains the identifier for the display that X11 programs should use by default.
HOME	Indicates the home directory of the current user: the default argument for the cd built-in command.
IFS	Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.
LANG	LANG expands to the default system locale; LC_ALL can be used to override this. For example, if its value is pt_BR, then the language is set to (Brazilian) Portuguese and the locale to Brazil.
LD_LIBRARY_PATH	On many Unix systems with a dynamic linker, contains a colon-separated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.
PATH	Indicates search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
PWD	Indicates the current working directory as set by the cd command.
RANDOM	Generates a random integer between 0 and 32,767 each time it is referenced.
SHLVL	Increments by one each time an instance of bash is started. This variable is useful for determining whether the built-in exit command ends the current session.
TERM	Refers to the display type
TZ	Refers to Time zone. It can take values like GMT, AST, etc.

UID	Expands to the numeric user ID of the current user, initialized at shell startup.

Following is the sample example showing few environment variables:

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo $TERM
xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

CHAPTER

6

Java Basic Utilities

So far you must have got some idea about Unix OS and nature of its basic commands. This tutorial would cover few very basic but important Unix utilities which you would use in your day to day life.

Printing Files:

Before you print a file on a UNIX system, you may want to reformat it to adjust the margins, highlight some words, and so on. Most files can also be printed without reformatting, but the raw printout may not look quite as nice.

Many versions of UNIX include two powerful text formatters, **nroff** and **troff**. They are not covered in this tutorial but you would quit a lot material on the net for these utilities.

The pr Command:

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

Here is the syntax of **pr** command:

```
pr option(s) filename(s)
```

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

Option	Description
-k	Produces k columns of output
-d	Double-spaces the output (not on all pr versions).
-h "header"	Takes the next item as a report header.
-t	Eliminates printing of header and top/bottom margins.
-l PAGE_LENGTH	Set the page length to PAGE_LENGTH (66) lines. Default number of lines of text 56.
-o MARGIN	Offset each line with MARGIN (zero) spaces.

-w PAGE_WIDTH	Set page width to PAGE_WIDTH (72) characters for multiple text-column output only.

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

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

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

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

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

The lp and lpr Commands:

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

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

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

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

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

The lpstat and lpq Commands:

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

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

```
$lpstat -o
laserp-573  john  128865  Nov 7  11:27  on laserp
```

```
laserp-574  grace  82744  Nov 7  11:28
laserp-575  john   23347  Nov 7  11:35
$
```

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

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

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

The cancel and lprm Commands:

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

```
$cancel laserp-575
request "laserp-575" cancelled
$
```

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

```
$cancel laserp
request "laserp-573" cancelled
$
```

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

```
$lprm 575
dfA575diamond dequeued
cfA575diamond dequeued
$
```

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

Sending Email:

You use the Unix mail command to send and receive mail. Here is the syntax to send an email:

```
$mail [-s subject] [-c cc-addr] [-b bcc-addr] to-addr
```

Here are important options related to mail command:

Option	Description
-s	Specify subject on command line.
-c	Send carbon copies to list of users. List should be a comma-separated list of names.

-b	Send blind carbon copies to list. List should be a comma-separated list of names.

Following is the example to send a test message to admin@yahoo.com.

```
$mail -s "Test Message" admin@yahoo.com
```

You are then expected to type in your message, followed by an "control-D" at the beginning of a line. To stop simply type dot (.) as follows:

```
Hi,

This is a test
.
Cc:
```

You can send a complete file using a redirect < operator as follows:

```
$mail -s "Report 05/06/07" admin@yahoo.com < demo.txt
```

To check incoming email at your Unix system you simply type email as follows:

```
$mail
no email
```

CHAPTER

7

Unix Pipes and Filters

You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.

To make a pipe, put a vertical bar (|) on the command line between two commands.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a *filter*.

The grep Command:

The grep program searches a file or files for lines that have a certain pattern. The syntax is:

```
$grep pattern file(s)
```

The name "grep" derives from the ed (a UNIX line editor) command g/re/p which means "globally search for a regular expression and print all lines containing it."

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with grep command:

Option	Description
-v	Print all lines that do not match pattern.

-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l")
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

Next, let's use a regular expression that tells grep to find lines with "carol", followed by zero or more other characters abbreviated in a regular expression as ".*"), then followed by "Aug".

Here we are using *-i* option to have case insensitive search:

```
$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

The sort Command:

The **sort** command arranges lines of text alphabetically or numerically. The example below sorts the lines in the food file:

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

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting:

Option	Description
-n	Sort numerically (example: 10 will sort after 2), ignore blanks and tabs.
-r	Reverse the order of sort.
-f	Sort upper- and lowercase together.
+x	Ignore first x fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by order of size.

The following pipe consists of the commands **ls, grep,** and **sort**:

```
$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by order of size, and prints them to the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

The pg and more Commands:

A long output would normally zip by you on the screen, but if you run text through more or pg as a filter, the display stops after each screenful of text.

Let's assume that you have a long directory listing. To make it easier to read the sorted listing, pipe the output through **more** as follows:

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc     14827 Aug  9 12:40 ch03
         .
         .
         .
-rw-rw-rw-  1 john  doc     16867 Aug  6 15:56 ch05
--More--(74%)
```

The screen will fill up with one screenful of text consisting of lines sorted by order of file size. At the bottom of the screen is the **more** prompt where you can type a command to move through the sorted text.

When you're done with this screen, you can use any of the commands listed in the discussion of the more program.

CHAPTER

8

Unix Processes Management

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the **ls**command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program. The operating system tracks processes through a five digit ID number known as the **pid** or process ID . Each process in the system has a unique pid.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any one time, no two processes with the same pid exist in the system because it is the pid that UNIX uses to track each process.

Starting a Process:

When you start a process (run a command), there are two ways you can run it:

- Foreground Processes
- Background Processes

Foreground Processes:

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If I want to list all the files in my current directory, I can use the following command:

```
$ls ch*.doc
```

This would display all the files whose name start with ch and ends with .doc:

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes:

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

```
$ls ch*.doc &
```

This would also display all the files whose name start with ch and ends with .doc:

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

Here if the **ls** command wants any input (which it does not), it goes into a stop state until I move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and process ID. You need to know the job number to manipulate it between background and foreground.

If you press the Enter key now, you see the following:

```
[1]   +   Done                 ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes:

It is easy to see your own processes by running the **ps** (process status) command as follows:

```
$ps
PID       TTY      TIME        CMD
18358     ttyp3    00:00:00    sh
18361     ttyp3    00:01:31    abiword
18789     ttyp3    00:00:00    ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example:

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
```

```
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fileds displayed by ps -f command:

Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

There are other options which can be used along with **ps** command:

Option	Description
-a	Shows information about all users
-x	Shows information about processes without terminals.
-u	Shows additional information like -f option.
-e	Display extended information.

Stopping Processes:

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

If a process is running in background mode then first you would need to get its Job ID using **ps**command and after that you can use **kill** command to kill the process as follows:

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here **kill** command would terminate first_one process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows:

```
$kill -9 6738
Terminated
```

Parent and Child Processes:

Each unix process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes:

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," **init** process, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie, or defunct, process. The process is dead and not being used. These processes are different from orphan processes.They are the processes that has completed execution but still has an entry in the process table.

Daemon Processes:

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon process has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

If you have a program which needs to do long processing then its worth to make it a daemon and run it in background.

The top Command:

The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Here is simple syntax to run top command and to see the statistics of CPU utilization by different processes:

```
$top
```

Job ID Versus Process ID:

Background and suspended processes are usually manipulated via job number (job ID). This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in series or at the same time, in parallel, so using the job ID is easier than tracking the individual processes.

CHAPTER

9

Unix Communication

When you work in a distributed environment then you need to communicate with remote users and you also need to access remote Unix machines.

There are several Unix utilities which are especially useful for users computing in a networked, distributed environment. This tutorial lists few of them:

The ping Utility:

The ping command sends an echo request to a host available on the network. Using this command you can check if your remote host is responding well or not.

The ping command is useful for the following:

- Tracking and isolating hardware and software problems.
- Determining the status of the network and various foreign hosts.
- Testing, measuring, and managing networks.

Syntax:

Following is the simple syntax to use **ping** command:

```
$ping hostname or ip-address
```

Above command would start printing a response after every second. To come out of the command you can terminate it by pressing CNTRL + C keys.

Example:

Following is the example to check the availability of a host available on the network:

```
$ping google.com
PING google.com (74.125.67.100) 56(84) bytes of data.
64 bytes from 74.125.67.100: icmp_seq=1 ttl=54 time=39.4 ms
64 bytes from 74.125.67.100: icmp_seq=2 ttl=54 time=39.9 ms
64 bytes from 74.125.67.100: icmp_seq=3 ttl=54 time=39.3 ms
```

```
64 bytes from 74.125.67.100: icmp_seq=4 ttl=54 time=39.1 ms
64 bytes from 74.125.67.100: icmp_seq=5 ttl=54 time=38.8 ms
--- google.com ping statistics ---
22 packets transmitted, 22 received, 0% packet loss, time 21017ms
rtt min/avg/max/mdev = 38.867/39.334/39.900/0.396 ms
$
```

If a host does not exist then it would behave something like this:

```
$ping giiiiiigle.com
ping: unknown host giiiiigle.com
$
```

The ftp Utility:

Here ftp stands for **F**ile **T**ransfer **P**rotocol. This utility helps you to upload and download your file from one computer to another computer.

The ftp utility has its own set of UNIX like commands which allow you to perform tasks such as:

- Connect and login to a remote host.
- Navigate directories.
- List directory contents
- Put and get files
- Transfer files as ascii, ebcdic or binary

Syntax:

Following is the simple syntax to use **ping** command:

```
$ftp hostname or ip-address
```

Above command would prompt you for login ID and password. Once you are authenticated, you would have access on the home directory of the login account and you would be able to perform various commands.

Few of the useful commands are listed below:

Command	Description
put filename	Upload filename from local machine to remote machine.
get filename	Download filename from remote machine to local machine.
mput file list	Upload more than one files from local machine to remove machine.
mget file list	Download more than one files from remote machine to local machine.
prompt off	Turns prompt off, by default you would be prompted to upload or download movies using mput or mget commands.

prompt on	Turns prompt on.
Dir	List all the files available in the current directory of remote machine.
cd dirname	Change directory to dirname on remote machine.
lcd dirname	Change directory to dirname on local machine.
Quit	Logout from the current login.

It should be noted that all the files would be downloaded or uploaded to or from current directories. If you want to upload your files in a particular directory then first you change to that directory and then upload required files.

Example:

Following is the example to show few commands:

```
$ftp amrood.com
Connected to amrood.com.
220 amrood.com FTP server (Ver 4.9 Thu Sep 2 20:35:07 CDT 2009)
Name (amrood.com:amrood): amrood
331 Password required for amrood.
Password:
230 User amrood logged in.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 1464
drwxr-sr-x   3 amrood   group       1024 Mar 11 20:04 Mail
drwxr-sr-x   2 amrood   group       1536 Mar  3 18:07 Misc
drwxr-sr-x   5 amrood   group        512 Dec  7 10:59 OldStuff
drwxr-sr-x   2 amrood   group       1024 Mar 11 15:24 bin
drwxr-sr-x   5 amrood   group       3072 Mar 13 16:10 mpl
-rw-r--r--   1 amrood   group     209671 Mar 15 10:57 myfile.out
drwxr-sr-x   3 amrood   group        512 Jan  5 13:32 public
drwxr-sr-x   3 amrood   group        512 Feb 10 10:17 pvm3
226 Transfer complete.
ftp> cd mpl
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 7320
-rw-r--r--   1 amrood   group       1630 Aug  8 1994  dboard.f
-rw-r-----   1 amrood   group       4340 Jul 17 1994  vttest.c
-rwxr-xr-x   1 amrood   group     525574 Feb 15 11:52 wave_shift
-rw-r--r--   1 amrood   group       1648 Aug  5 1994  wide.list
-rwxr-xr-x   1 amrood   group       4019 Feb 14 16:26 fix.c
226 Transfer complete.
ftp> get wave_shift
200 PORT command successful.
150 Opening data connection for wave_shift (525574 bytes).
226 Transfer complete.
528454 bytes received in 1.296 seconds (398.1 Kbytes/s)
ftp> quit
221 Goodbye.
```

```
$
```

The telnet Utility:

Many times you would be in need to connect to a remote Unix machine and work on that machine remotely. Telnet is a utility that allows a computer user at one site to make a connection, login and then conduct work on a computer at another site.

Once you are login using telnet, you can perform all the activities on your remotely connect machine. Here is example telnet session:

```
C:>telnet amrood.com
Trying...
Connected to amrood.com.
Escape character is '^]'.

login: amrood
amrood's Password:
*****************************************************
*                                                   *
*                                                   *
*    WELCOME TO AMROOD.COM                          *
*                                                   *
*                                                   *
*****************************************************

Last unsuccessful login: Fri Mar  3 12:01:09 IST 2009
Last login: Wed Mar  8 18:33:27 IST 2009 on pts/10

   {  do your work }

$ logout
Connection closed.
C:>
```

The finger Utility:

The finger command displays information about users on a given host. The host can be either local or remote.

Finger may be disabled on other systems for security reasons.

Following are the simple syntax to use finger command:

Check all the logged in users on local machine as follows:

```
$ finger
Login     Name       Tty      Idle  Login Time   Office
amrood               pts/0          Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on local machine:

```
$ finger amrood
Login: amrood                           Name: (null)
Directory: /home/amrood                 Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
```

```
No Plan.
```

Check all the logged in users on remote machine as follows:

```
$ finger @avtar.com
Login     Name       Tty      Idle  Login Time   Office
amrood               pts/0          Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on remote machine:

```
$ finger amrood@avtar.com
Login: amrood                           Name: (null)
Directory: /home/amrood                 Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

CHAPTER

10

Unix – The vi Editor

There are many ways to edit files in Unix and for me one of the best ways is using screen-oriented text editor **vi**. This editor enable you to edit lines in context with other lines in the file.
Now a days you would find an improved version of vi editor which is called **VIM**. Here VIM stands for **Vi**l**M**proved.

The vi is generally considered the de facto standard in Unix editors because:

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user friendly than any other editors like ed or ex.

You can use **vi** editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

Starting the vi Editor:

There are following way you can start using vi editor:

Command	Description
vi filename	Creates a new file if it already does not exist, otherwise opens existing file.
vi -R filename	Opens an existing file in read only mode.
view filename	Opens an existing file in read only mode.

Following is the example to create a new file **testfile** if it already does not exist in the current working directory:

```
$vi testfile
```

As a result you would see a screen something like as follows:

```
|
~
~
~
~
```

```
~
~
~
~
~
~
~
~
"testfile" [New File]
```

You will notice a tilde (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other nonviewable character present.

So now you have opened one file to start with. Before proceeding further let us understanding few minor but important concepts explained below.

Operation Modes:

While working with vi editor you would come across following two modes:

1. **Command mode:** This mode enables you to perform administrative tasks such as saving files, executing commands, moving the cursor, cutting (yanking) and pasting lines or words, and finding and replacing. In this mode, whatever you type is interpreted as a command.
2. **Insert mode:** This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and finally it is put in the file .

The vi always starts in command mode. To enter text, you must be in insert mode. To come in insert mode you simply type **i**. To get out of insert mode, press the **Esc** key, which will put you back into command mode.
Hint: If you are not sure which mode you are in, press the Esc key twice, and then you'll be in command mode. You open a file using vi editor and start type some characters and then come in command mode to understand the difference.

Getting Out of vi:

The command to quit out of vi is :q. Once in command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is **:q!**. This lets you exit vi without saving any of the changes.
The command to save the contents of the editor is **:w**. You can combine the above command with the quit command, or :wq and return.
The easiest way to save your changes and exit out of vi is the **ZZ** command. When you are in command mode, type ZZ and it will do the equivalent of :wq.
You can specify a different file name to save to by specifying the name after the :w. For example, if you wanted to save the file you were working as another filename called filename2, you would type **:w filename2** and return. Try it once.

Moving within a File:

To move around within a file without affecting your text, you must be in command mode (press Esc twice). Here are some of the commands you can use to move around one character at a time:

Command	Description
K	Moves the cursor up one line.
J	Moves the cursor down one line.
H	Moves the cursor to the left one character position.
L	Moves the cursor to the right one character position.

There are following two important points to be noted:

- The vi is case-sensitive, so you need to pay special attention to capitalization when using commands.
- Most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2j moves cursor two lines down the cursor location.

There are many other ways to move within a file in vi. Remember that you must be in command mode (press Esc twice). Here are some more commands you can use to move around the file:

Command	Description
0 or |	Positions cursor at beginning of line.
$	Positions cursor at end of line.
w	Positions cursor to the next word.
b	Positions cursor to previous word.
(	Positions cursor to beginning of current sentence.
)	Positions cursor to beginning of next sentence.
E	Move to the end of Blank delimited word
{	Move a paragraph back
}	Move a paragraph forward
[[	Move a section back
]]	Move a section forward
n|	Moves to the column n in the current line
1G	Move to the first line of the file
G	Move to the last line of the file
nG	Move to nth line of the file
:n	Move to nth line of the file
fc	Move forward to c
Fc	Move back to c
H	Move to top of screen
nH	Moves to nth line from the top of the screen
M	Move to middle of screen
L	Move to botton of screen
nL	Moves to nth line from the bottom of the screen
:x	Colon followed by a number would position the cursor on line number represented by **x**

Control Commands:

There are following useful command which you can use along with Control Key:

Command	Description
CTRL+d	Move forward 1/2 screen
CTRL+d	Move forward 1/2 screen
CTRL+f	Move forward one full screen
CTRL+u	Move backward 1/2 screen
CTRL+b	Move backward one full screen
CTRL+e	Moves screen up one line
CTRL+y	Moves screen down one line
CTRL+u	Moves screen up 1/2 page
CTRL+d	Moves screen down 1/2 page
CTRL+b	Moves screen up one page
CTRL+f	Moves screen down one page
CTRL+l	Redraws screen

Editing Files:

To edit the file, you need to be in the insert mode. There are many ways to enter insert mode from the command mode:

Command	Description
i	Inserts text before current cursor location.
I	Inserts text at beginning of current line.
a	Inserts text after current cursor location.
A	Inserts text at end of current line.
o	Creates a new line for text entry below cursor location.
O	Creates a new line for text entry above cursor location.

Deleting Characters:

Here is the list of important commands which can be used to delete characters and lines in an opened file:

Command	Description
x	Deletes the character under the cursor location.
X	Deletes the character before the cursor location.
dw	Deletes from the current cursor location to the next word.
d^	Deletes from current cursor position to the beginning of the line.
d$	Deletes from current cursor position to the end of the line.
D	Deletes from the cursor position to the end of the current line.

dd	Deletes the line the cursor is on.

As mentioned above, most commands in vi can be prefaced by the number of times you want the action to occur. For example, **2x** deletes two character under the cursor location and 2dd deletes two lines the cursor is on.

I would highly recommend to exercise all the above commands properly before proceeding further.

Change Commands:

You also have the capability to change characters, words, or lines in vi without deleting them. Here are the relevant commands:

Command	Description
cc	Removes contents of the line, leaving you in insert mode.
cw	Changes the word the cursor is on from the cursor to the lowercase w end of the word.
r	Replaces the character under the cursor. vi returns to command mode after the replacement is entered.
R	Overwrites multiple characters beginning with the character currently under the cursor. You must use **Esc** to stop the overwriting.
s	Replaces the current character with the character you type. Afterward, you are left in insert mode.
S	Deletes the line the cursor is on and replaces with new text. After the new text is entered, vi remains in insert mode.

Copy and Past Commands:

You can copy lines or words from one place and then you can past them at another place using following commands:

Command	Description
Yy	Copies the current line.
Yw	Copies the current word from the character the lowercase w cursor is on until the end of the word.
P	Puts the copied text after the cursor.
P	Puts the yanked text before the cursor.

Advanced Commands:

There are some advanced commands that simplify day-to-day editing and allow for more efficient use of vi:

Command	Description
J	Join the current line with the next one. A count joins that many lines.
<<	Shifts the current line to the left by one shift width.
>>	Shifts the current line to the right by one shift width.
~	Switch the case of the character under the cursor.

^G	Press CNTRL and G keys at the same time to show the current filename and the status.
U	Restore the current line to the state it was in before the cursor entered the line.
U	Undo the last change to the file. Typing 'u' again will re-do the change.
J	Join the current line with the next one. A count joins that many lines.
:f	Displays current position in the file in % and file name, total number of file.
:f filename	Renames current file to filename.
:w filename	Write to file filename.
:e filename	Opens another file with filename.
:cd dirname	Changes current working directory to dirname.
:e #	Use to toggle between two opened files.
:n	In case you open multiple files using vi, use :n to go to next file in the series.
:p	In case you open multiple files using vi, use :p to go to previous file in the series.
:N	In case you open multiple files using vi, use :N to go to previous file in the series.
:r file	Reads file and inserts it after current line
:nr file	Reads file and inserts it after line n.

Word and Character Searching:

The vi editor has two kinds of searches: string and character. For a string search, the / and ? commands are used. When you start these commands, the command just typed will be shown on the bottom line, where you type the particular string to look for.

These two commands differ only in the direction where the search takes place:

- The / command searches forwards (downwards) in the file.
- The ? command searches backwards (upwards) in the file.

The n and N commands repeat the previous search command in the same or opposite direction, respectively. Some characters have special meanings while using in search command and preceded by a backslash (\) to be included as part of the search expression.

Character	Description
^	Search at the beginning of the line. (Use at the beginning of a search expression.)
.	Matches a single character.
*****	Matches zero or more of the previous character.
$	End of the line (Use at the end of the search expression.)
[	Starts a set of matching, or non-matching expressions.
<	Put in an expression escaped with the backslash to find the ending or beginning

	of a word.
>	See the '<' character description above.

The character search searches within one line to find a character entered after the command. The f and F commands search for a character on the current line only. f searches forwards and F searches backwards and the cursor moves to the position of the found character.

The t and T commands search for a character on the current line only, but for t, the cursor moves to the position before the character, and T searches the line backwards to the position after the character.

Set Commands:

You can change the look and feel of your vi screen using the following **:set** commands. To use these commands you have to come in command mode then type **:set** followed by any of the following options:

Command	Description
:set ic	Ignores case when searching
:set ai	Sets autoindent
:set noai	To unset autoindent.
:set nu	Displays lines with line numbers on the left side.
:set sw	Sets the width of a software tabstop. For example you would set a shift width of 4 with this command: **:set sw=4**
:set ws	If *wrapscan* is set, if the word is not found at the bottom of the file, it will try to search for it at the beginning.
:set wm	If this option has a value greater than zero, the editor will automatically "word wrap". For example, to set the wrap margin to two characters, you would type this: **:set wm=2**
:set ro	Changes file type to "read only"
:set term	Prints terminal type
:set bf	Discards control characters from input

Running Commands:

The vi has the capability to run commands from within the editor. To run a command, you only need to go into command mode and type **:!** command.
For example, if you want to check whether a file exists before you try to save your file to that filename, you can type **:! ls** and you will see the output of ls on the screen.

When you press any key (or the command's escape sequence), you are returned to your vi session.

Replacing Text:

The substitution command (**:s/**) enables you to quickly replace words or groups of words within your files. Here is the simple syntax:

```
:s/search/replace/g
```

The g stands for globally. The result of this command is that all occurrences on the cursor's line are changed.

IMPORTANT:

Here are the key points to your success with vi:

- You must be in command mode to use commands. (Press Esc twice at any time to ensure that you are in command mode.)
- You must be careful to use the proper case (capitalization) for all commands.
- You must be in insert mode to enter text.

CHAPTER

11

Unix- What is Shell

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt:

The prompt, $, which is called command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

The shell reads your input after you press Enter. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of **date** command which displays current date and time:

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using environment variable PS1 explained in Environment tutorial.

Shell Types:

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)

- POSIX shell (sh)

The different C-type shells follow:

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.

In this tutorial, we are going to cover most of the Shell concepts based on Borne Shell.

Shell Scripts:

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign, #, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

Example Script:

Assume we create a test.sh script. Note all the scripts would have **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example:

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

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.*

To create a script containing these commands, you put the shebang line first and then add the commands:

```
#!/bin/bash
pwd
ls
```

Shell Comments:

You can put your comments in your script as follows:

```
#!/bin/bash
```

```
# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Now you save the above content and make this script executable as follows:

```
$chmod +x test.sh
```

Now you have your shell script ready to be executed as follows:

```
$./test.sh
```

This would produce following result:

```
/home/amrood
index.htm  unix-basic_utilities.htm  unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

Note: To execute your any program available in current directory you would execute using**./program_name**

Extended Shell Scripts:

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is sample run of the script:

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

CHAPTER

12

Unix- Using Variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix Shell variables would have their names in UPPERCASE.

The following examples are valid variable names:

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names:

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.

Defining Variables:

Variables are defined as follows:

```
variable_name=variable_value
```

For example:

```
NAME="Zara Ali"
```

Above example defines the variable NAME and assigns it the value "Zara Ali". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The shell enables you to store any value you want in a variable. For example:

```
VAR1="Zara Ali"
VAR2=100
```

Accessing Values:

To access the value stored in a variable, prefix its name with the dollar sign ($):

For example, following script would access the value of defined variable NAME and would print it on STDOUT:

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

This would produce following value:

```
Zara Ali
```

Read-only Variables:

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME:

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

This would produce following result:

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables:

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command:

```
unset variable_name
```

Above command would unset the value of a defined variable. Here is a simple example:

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

```
NAME="Zara Ali"
unset NAME
echo $NAME
```

Above example would not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types:

When a shell is running, three main types of variables are present:

- **Local Variables:** A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
- **Environment Variables:** An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables:** A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

CHAPTER

13

Unix-Special Variables

Previous tutorials warned about using certain nonalphanumeric characters in your variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the $ character represents the process ID number, or PID, of the current shell:

```
$echo $$
```

Above command would write PID of the current shell:

```
29949
```

The following table shows a number of special variables that you can use in your shell scripts:

Variable	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Command-Line Arguments:

The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to command line:

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script:

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
First Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@:

There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameter specifies all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters:

```
#!/bin/sh

for TOKEN in $*
do
   echo $TOKEN
done
```

There is one sample run for the above script:

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years
Old
```

Note: Here **do**...**done** is a kind of loop which we would cover in subsequent tutorial.

Exit Status:

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command:

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
First Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

CHAPTER

14

Unix – Using Arrays

A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.

Shell supports a different type of variable called an array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

Defining Array Values:

The difference between an array variable and a scalar variable can be explained as follows.

Say that you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows:

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

```
array_name[index]=value
```

Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands:

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If you are using **ksh** shell the here is the syntax of array initialization:

```
set -A array_name value1 value2 ... valuen
```

If you are using **bash** shell the here is the syntax of array initialization:

```
array_name=(value1 ... valuen)
```

Accessing Array Values:

After you have set any array variable, you access it as follows:

```
${array_name[index]}
```

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is the simplest example:

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

This would produce following result:

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

You can access all the items in an array in one of the following ways:

```
${array_name[*]}
${array_name[@]}
```

Here array_name is the name of the array you are interested in. Following is the simplest example:

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

This would produce following result:

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

CHAPTER

15

Unix - Basic Operators

There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

There are following operators which we are going to discuss:

- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers:

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result:

```
Total value : 4
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.

Arithmetic Operators:

There are following arithmetic operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a * $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

Example:

Here is an example which uses all the arithmetic operators:

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"
```

```
val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

This would produce following result:

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.
- You should use \ on the * symbol for multiplication.
- **if...then...fi** statement is a decision making statement which has been explained in next chapter.

Relational Operators:

Bourne Shell supports following relational operators which are specific to numeric values. These operators would not work for string values unless their value is numeric.

For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.

-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

Example:

Here is an example which uses all the relational operators:

```
#!/bin/sh

a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
```

```
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

This would produce following result:

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision making statement which has been explained in next chapter.

Boolean Operators:

There are following boolean operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

Example:

Here is an example which uses all the boolean operators:

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

```
a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

This would produce following result:

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision making statement which has been explained in next chapter.

String Operators:

There are following string operators supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then:

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.

!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.
Str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

Example:

Here is an example which uses all the string operators:

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

This would produce following result:

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.

- **if...then...else...fi** statement is a decision making statement which has been explained in next chapter.

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:

Here is an example which uses all the file test operators:

Assume a variable file holds an existing file name "/var/www/tutorialspoint/unix/test.sh" whose size is 100 bytes and has read, write and execute permission on:

```
#!/bin/sh

file="/var/www/tutorialspoint/unix/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
   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
```

This would produce following result:

```
File has read access
File has write permission
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision making statement which has been explained in next chapter.

C Shell Operators:

This lists down all the operators available in C Shell. Here most of the operators are very similar to what we have in C Programming language.

Operators are listed in order of decreasing precedence:

Arithmetic and Logical Operators:

Operator	Description
( )	Change precedence
~	1's complement
!	Logical negation
*	Multiply
/	Divide
%	Modulo
+	Add
-	Subtract
<<	Left shift

>>	Right shift
==	String comparison for equality
!=	String comparison for non equality
=~	Pattern matching
&	Bitwise "and"
^	Bitwise "exclusive or"
|	Bitwise "inclusive or"
&&	Logical "and"
||	Logical "or"
++	Increment
--	Decrement
=	Assignment
*=	Multiply left side by right side and update left side
/=	Divide left side by right side and update left side
+=	Add left side to right side and update left side
-=	Subtract left side from right side and update left side
^=	"Exclusive or" left side to right side and update left side
%=	Divide left by right side and update left side with remainder

File Test Operators:

There are following operators to test various properties associated with a Unix file.

Operator	Description
-r file	Checks if file is readable if yes then condition becomes true.
-w file	Check if file is writable if yes then condition becomes true.
-x file	Check if file is execute if yes then condition becomes true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.
-z file	Check if file has size greater than 0 if yes then condition becomes true.
-d file	Check if file is a directory if yes then condition becomes true.
-e file	Check if file exists. Is true even if file is a directory but exists.
-o file	Check if user owns the file. It returns true if user is the owner of the file.

Korn Shell Operators:

This lists down all the operators available in Korn Shell. Here most of the operators are very similar to what we have in C Programming language.

Operators are listed in order of decreasing precedence:

Arithmetic and Logical Operators:

Operator	Description
+	Unary plus
-	Unary minus
!~	Logical negation; binary inversion (one's complement)
*	Multiply
/	Divide
%	Modulo
+	Add
-	Subtract
<<	Left shift
>>	Right shift
==	String comparison for equality
!=	String comparison for non equality
=~	Pattern matching
&	Bitwise "and"
^	Bitwise "exclusive or"
|	Bitwise "inclusive or"
&&	Logical "and"
||	Logical "or"
++	Increment
--	Decrement
=	Assignment

File Test Operators:

There are following operators to test various properties associated with a Unix file.

Operator	Description

-r file	Checks if file is readable if yes then condition becomes true.
-w file	Check if file is writable if yes then condition becomes true.
-x file	Check if file is execute if yes then condition becomes true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.
-d file	Check if file is a directory if yes then condition becomes true.
-e file	Check if file exists. Is true even if file is a directory but exists.

CHAPTER

16

Unix – Decision Making

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements:

- The **if...else** statements
- The **case...esac** statement

The if...else statements:

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement:

if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error.

If **expression** is a shell command then it would be assumed true if it return 0 after its execution. If it is a boolean expression then it would be true if it returns true.

Example:

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

Example:

If we take above example then it can be written in better way using *if...else* statement as follows:

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

if...elif...else...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax:

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

There is nothing special about this code. It is just a series of *if* statements, where each *if* is part of the*else* clause of the previous statement. Here statement(s) are executed based on the true condition, if non of the condition is true then *else* block is executed.

Example:

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

This will produce following result:

```
a is less than b
```

Most of the if statements check relations using relational operators discussed in previous chapter.

The case...esac Statement:

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which is detailed here:

case...esac statement

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Shell support **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Syntax:

The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example:

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

This will produce following result:

```
New Zealand is famous for kiwi.
```

A good use for a case statement is the evaluation of command line arguments as follows:

```
#!/bin/sh

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Here is a sample run of this program:

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++ and PERL etc.

CHAPTER

17

Unix – Shell Loops

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers:

The while loop

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax:

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement would be not executed and program would jump to the next line after done statement.

Example:

Here is a simple example that uses the while loop to display the numbers zero to nine:

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result:

```
0
1
2
3
4
```

```
5
6
7
8
9
```

Each time this loop executes, the variable a is checked to see whether it has a value that is less than 10. If the value of a is less than 10, this test condition has an exit status of 0. In this case, the current value of a is displayed and then a is incremented by 1.

The for loop

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

Syntax:

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example:

Here is a simple example that uses for loop to span through the given list of numbers:

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

This will produce following result:

```
0
1
2
3
4
5
6
7
8
9
```

Following is the example to display all the files starting with **.bash** and available in your home. I'm executing this script from my root:

```
#!/bin/sh

for FILE in $HOME/.bash*
do
   echo $FILE
```

```
done
```

This will produce following result:

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

The until loop

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax:

```
until command

do

   Statement(s) to be executed until command is true

done
```

Here Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If*command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

Example:

Here is a simple example that uses the until loop to display the numbers zero to nine:

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]

do

   echo $a

   a=`expr $a + 1`

done
```

This will produce following result:

```
0

1

2

3

4
```

```
5

6

7

8

9
```

The select loop

The *select* loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax:

```
select var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

For every selection a set of commands would be executed with-in the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

Example:

Here is a simple example to let the user select a drink of choice:

```
#!/bin/ksh

select DRINK in tea cofee water juice appe all none
do
   case $DRINK in
      tea|cofee|water|all)
         echo "Go to canteen"
         ;;
      juice|appe)
         echo "Available at home"
      ;;
      none)
         break
      ;;
      *) echo "ERROR: Invalid selection"
      ;;
   esac
done
```

The menu presented by the select loop looks like the following:

```
$./test.sh
1) tea
2) cofee
3) water
```

```
4) juice
5) appe
6) all
7) none
#? juice
Available at home
#? none
$
```

You can change the prompt displayed by the select loop by altering the variable PS3 as follows:

```
$PS3="Please make a selection => " ; export PS3
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
Please make a selection => juice
Available at home
Please make a selection => none
$
```

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

Once you have good programming practice you would start using appropriate loop based on situation. Here while and for loops are available in most of the other programming languages like C, C++ and PERL etc.

Nesting Loops:

All the loops support nesting concept which means you can put one loop inside another similar or different loops. This nesting can go upto unlimited number of times based on your requirement.

Here is an example of nesting **while** loop and similar way other loops can be nested based on programming requirement:

Nesting while Loops:

It is possible to use a while loop as part of the body of another while loop.

Syntax:

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done

   Statement(s) to be executed if command1 is true
done
```

Example:

Here is a simple example of loop nesting, let's add another countdown loop inside the loop that you used to count to nine:

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

This will produce following result. It is important to note how **echo -n** works here. Here **-n** option let echo to avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

CHAPTER

18

Unix – Loop Control

So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this tutorial you will learn following two statements used to control shell loops:

1. The **break** statement
2. The **continue** statement

The infinite Loop:

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

Example:

Here is a simple example that uses the while loop to display the numbers zero to nine:

```
#!/bin/sh

a=10

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This loop would continue forever because a is alway greater than 10 and it would never become less than 10. So this true example of infinite loop.

The break statement:

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax:

The following **break** statement would be used to come out of a loop:

```
break
```

The break command can also be used to exit from a nested loop using this format:

```
break n
```

Here **n** specifies the nth enclosing loop to exit from.

Example:

Here is a simple example which shows that loop would terminate as soon as a becomes 5:

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

This will produce following result:

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0:

```
#!/bin/sh

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

This will produce following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from inner loop as well.

```
1 0
1 5
```

The continue statement:

The **continue** statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax:

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example:

The following loop makes use of continue statement which returns from the continue statement and start processing next statement:

```
#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
   echo "Found odd number"
done
```

This will produce following result:

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

CHAPTER

19

Unix – Shell Substitutions

What is Substitution?

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example:

Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line:

```
#!/bin/sh

a=10
echo -e "Value of a is $a \n"
```

This would produce following result. Here **-e** option enables interpretation of backslash escapes.

```
Value of a is 10
```

Here is the result without -e option:

```
Value of a is 10\n
```

Here are following escape sequences which can be used in echo command:

Escape	Description
\\	Backslash
\a	alert (BEL)
\b	Backspace
\c	suppress trailing newline
\f	form feed

\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

You can use **-E** option to disable interpretation of backslash escapes (default).

You can use **-n** option to disable insertion of new line.

Command Substitution:

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax:

The command substitution is performed when a command is given as:

```
`command`
```

When performing command substitution make sure that you are using the backquote, not the single quote character.

Example:

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution:

```
#!/bin/sh

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

This will produce following result:

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

Variable Substitution:

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions:

Form	Description
${var}	Substitue the value of *var*.
${var:-word}	If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
${var:=word}	If *var* is null or unset, *var* is set to the value of **word**.
${var:?message}	If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
${var:+word}	If *var* is set, *word* is substituted for var. The value of *var* does not change.

Example:

Following is the example to show various states of the above substitution:

```
#!/bin/sh

echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

This would produce following result:

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

CHAPTER

20

Unix – Quoting Mechanisms

The Metacharacters

Unix shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example **?** matches with a single charater while listing files in a directory and an * would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters):

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab
```

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

Example:

Following is the example which show how to print a * or a?:

```
#!/bin/sh

echo Hello; Word
```

This would produce following result.

```
Hello
./test.sh: line 2: Word: command not found

shell returned 127
```

Now let us try using a quoted character:

```
#!/bin/sh

echo Hello\; Word
```

This would produce following result:

```
Hello; Word
```

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell:

```
#!/bin/sh

echo "I have \$1200"
```

This would produce following result:

```
I have $1200
```

There are following four forms of quotings:

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions: $ ` \$ \' \" \\
Backslash	Any character immediately following the backslash loses its special meaning.
Back Quote	Anything in between back quotes would be treated as a command and would be executed.

The Single Quotes:

Consider an echo command that contains many special shell characters:

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read:

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string:

```
echo '<-$1500.**>; (update?) [y|n]'
```

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you would preceed that using a backslash (\) as follows:

```
echo 'It\'s Shell Programming'
```

The Double Quotes:

Try to execute the following shell script. This shell script makes use of single quote:

```
VAR=ZARA
echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

This would produce following result:

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows:

```
VAR=ZARA
echo "$VAR owes <-\$1500.**>; [ as of (`date +%m/%d`) ]"
```

Now this would produce following result:

```
ZARA owes <-$1500.**>; [ as of (07/02) ]
```

Double quotes take away the special meaning of all characters except the following:

- $ for parameter substitution.
- Backquotes for command substitution.
- \$ to enable literal dollar signs.
- \` to enable literal backquotes.
- \" to enable embedded double quotes.
- \\ to enable embedded backslashes.
- All other \ characters are literal (not special).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows:

```
echo 'It\'s Shell Programming'
```

The Back Quotes:

Putting any Shell command in between back quotes would execute the command

Syntax:

Here is the simple syntax to put any Shell **command** in between back quotes:

Example:

```
var=`command`
```

Example:

Following would execute **date** command and produced result would be stored in DATA variable.

```
DATE=`date`

echo "Current Date: $DATE"
```

This would produce following result:

```
Current Date: Thu Jul  2 05:28:45 MST 2009
```

CHAPTER

21

Unix – IO Redirections

Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is also your terminal by default.

Output Redirection:

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal:

Check following **who** command which would redirect complete output of the command in users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check *users* file then it would have complete content:

```
$ cat users
oko          tty01    Sep 12 07:30
ai           tty15    Sep 12 13:32
ruth         tty21    Sep 12 10:10
pat          tty24    Sep 12 13:07
steve        tty25    Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example:

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows:

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection:

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows:

```
$ wc -l users
2 users
$
```

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file *users*:

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Here Document:

A *here document* is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is:

```
command << delimiter
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count total number of line:

```
$wc -l << EOF
```

```
        This is a simple lookup program
        for good (and bad) restaurants
        in Cape Town.
EOF
3
$
```

You can use *here document* to print multiple lines using your script as follows:

```
#!/bin/sh

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

This would produce following result:

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

The following script runs a session with the vi text editor and save the input in the file test.txt.

```
#!/bin/sh

filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following:

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt:

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

Discard the output:

Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file /dev/null:

```
$ command > /dev/null
```

Here command is the name of the command you want to execute. The file /dev/null is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect STDERR to STDOUT:

```
$ command > /dev/null 2>&1
```

Here 2 represents STDERR and 1 represents STDOUT. You can display a message on to STDERR by redirecting STDIN into STDERR as follows:

```
$ echo message 1>&2
```

Redirection Commands:

Following is the complete list of commands which you can use for redirection:

Command	Description
pgm > file	Output of pgm is redirected to file
pgm < file	Program pgm reads its input from file.
pgm >> file	Output of pgm is appended to file.
n > file	Output from stream with descriptor n redirected to file.
n >> file	Output from stream with descriptor n appended to file.
n >& m	Merge output from stream n with stream m.
n <& m	Merge input from stream n with stream m.
<< tag	Standard input comes from here through next tag at start of line.
|	Takes output from one program, or process, and sends it to another.

Note that file descriptor 0 is normally standard input (STDIN), 1 is standard output (STDOUT), and 2 is standard error output (STDERR).

CHAPTER

22

Unix – Shell Functions

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions:

To declare a function, simply use the following syntax:

```
function_name () {
   list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

Following is the simple example of using function:

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

When you would execute above script it would produce following result:

```
$./test.sh
Hello World
$
```

Pass Parameters to a Function:

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
$
```

Returning Values from Functions:

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the **return** command whose syntax is as follows:

```
return code
```

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example:

Following function returns a value 1:

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World $1 $2"
   return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returnd by last command
ret=$?
```

```
echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
Return value is 10
$
```

Nested Functions:

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a *recursive function*.

Following simple example demonstrates a nesting of two functions:

```
#!/bin/sh

# Calling one function from another
number_one () {
   echo "This is the first function speaking..."
   number_two
}

number_two () {
   echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...
This is now the second function speaking...
```

Function Call from Prompt:

You can put definitions for commonly used functions inside your *.profile* so that they'll be available whenever you log in and you can use them at command prompt.

Alternatively, you can group the definitions in a file, say *test.sh*, and then execute the file in the current shell by typing:

```
$. test.sh
```

This has the effect of causing any functions defined inside test.sh to be read in and defined to the current shell as follows:

```
$ number_one
This is the first function speaking...
This is now the second function speaking...
$
```

To remove the definition of a function from the shell, you use the unset command with the .f option. This is the same command you use to remove the definition of a variable to the shell.

```
$unset .f function_name
```

CHAPTER

23

Unix - Manpage Help

All the Unix commands come with a number of optional and mandatory options. It is very common to forget complete syntax of these commands.

Because no one can possibly remember every Unix command and all its options, there has been online help available since Unix's earliest days.

Unix's version of help files are called **man pages**. If you know any command name but you do not know how to use it, then Man Pages are here to help you at every step.

Syntax:

Here is the simple command to get the detail of any Unix command while working with the system:

```
$man command
```

Example:

Now you imagine any command for which you want to get help. Assuming you want to know about **pwd**then you simply need to use the following command:

```
$man pwd
```

The above command would open a help for you which would give you complete information about **pwd**command. Try it yourself at your command prompt to get more detail on

You can get complete detail on **man** command itself using the following command:

```
$man man
```

Man Page Sections:

Man pages are generally divided into sections, which generally vary by the man page author's preference. Here are some of the more common sections:

Section	Description

NAME	Name of the command
SYNOPSIS	General usage parameters of the command.
DESCRIPTION	Generally describes of the command and what it does
OPTIONS	Describes all the arguments or options to the command
SEE ALSO	Lists other commands that are directly related to the command in the man page or closely resembling its functionality.
BUGS	Explains any known issues or bugs that exist with the command or its output
EXAMPLES	Common usage examples that give the reader an idea of how the command can be used.
AUTHORS	The author of the man page/command.

So finally, I would say that man pages are a vital resource and the first avenue of research when you need information about commands or files in a Unix system.

Useful Shell Commands:

Now you know how to proceed, following link would give you a list of most important and very frequently used Unix Shell commands.

If you do not know how to use any command then use man page to get complete detail about the command.

Here is the list of Unix Shell - Useful Commands

CHAPTER

24

Unix - Regular Expressions

A regular expression is a string that can be used to describe several sequences of characters. Regular expressions are used by several different Unix commands, including **ed, sed, awk, grep,** and, to a more limited extent, **vi**.

This tutorial would teach you how to use regular expression along with **sed**.
Here sed stands for **s**tream **ed**itor is a stream oriented editor which was created exclusively for executing scripts. Thus all the input you feed into it passes through and goes to STDOUT and it does not change the input file.

Invoking sed:

Before we start, let us take make sure you have a local copy of /etc/passwd text file to work with **sed**.

As mentioned previously, sed can be invoked by sending data through a pipe to it as follows:

```
$ cat /etc/passwd | sed
Usage: sed [OPTION]... {script-other-script} [input-file]...

  -n, --quiet, --silent
                 suppress automatic printing of pattern space
  -e script, --expression=script
.............................
```

The cat command dumps the contents of /etc/passwd to sed through the pipe into sed's pattern space. The pattern space is the internal work buffer that sed uses to do its work.

The sed General Syntax:

Following is the general syntax for sed

```
/pattern/action
```

Here, **pattern** is a regular expression, and **action** is one of the commands given in the following table. If **pattern** is omitted, **action** is performed for every line as we have seen above.

The slash characters (/) that surround the pattern are required because they are used as delimiters.

Range	Description
p	Prints the line

d	Deletes the line
s/pattern1/pattern2/	Substitutes the first occurrence of pattern1 with pattern2.

Deleting All Lines with sed:

Invoke sed again, but this time tell sed to use the editing command delete line, denoted by the single letter d:

```
$ cat /etc/passwd | sed 'd'
$
```

Instead of invoking sed by sending a file to it through a pipe, you can instruct sed to read the data from a file, as in the following example.

The following command does exactly the same thing as the previous Try It Out, without the cat command:

```
$ sed -e 'd' /etc/passwd
$
```

The sed Addresses:

Sed also understands something called addresses. Addresses are either particular locations in a file or a range where a particular editing command should be applied. When sed encounters no addresses, it performs its operations on every line in the file.

The following command adds a basic address to the sed command you've been using:

```
$ cat /etc/passwd | sed '1d' |more
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
$
```

Notice that the number 1 is added before the delete edit command. This tells sed to perform the editing command on the first line of the file. In this example, sed will delete the first line of /etc/password and print the rest of the file.

The sed Address Ranges:

So what if you want to remove more than one line from a file? You can specify an address range with sed as follows:

```
$ cat /etc/passwd | sed '1, 5d' |more
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
$
```

Above command would be applied on all the lines starting from 1 through 5. So it deleted first five lines.

Try out the following address ranges:

Range	Description
'4,10d'	Lines starting from 4th till 10th are deleted
'10,4d'	Only 10th line is deleted, because sed does not work in reverse direction.
'4,+5d'	This will match line 4 in the file, delete that line, continue to delete the next five lines, and then cease its deletion and print the rest
'2,5!d'	This will deleted everything except starting from 2nd till 5th line.
'1~3d'	This deletes the first line, steps over the next three lines, and then deletes the fourth line. Sed continues applying this pattern until the end of the file.
'2~2d'	This tells sed to delete the second line, step over the next line, delete the next line, and repeat until the end of the file is reached.
'4,10p'	Lines starting from 4th till 10th are printed
'4,d'	This would generate syntax error.
',10d'	This would also generate syntax error.

Note: While using **p** action, you should use **-n** option to avoid repetition of line printing. Check the difference in between following two commands:

```
$ cat /etc/passwd | sed -n '1,3p'
```

Check the above command without **-n** as follows:

```
$ cat /etc/passwd | sed '1,3p'
```

The Substitution Command:

The substitution command, denoted by **s**, will substitute any string that you specify with any other string that you specify.

To substitute one string with another, you need to have some way of telling sed where your first string ends and the substitution string begins. This is traditionally done by bookending the two strings with the forward slash (/) character.

The following command substitutes the first occurrence on a line of the string **root** with the string**amrood**.

```
$ cat /etc/passwd | sed 's/root/amrood/'
amrood:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
.........................
```

It is very important to note that sed substitutes only the first occurrence on a line. If the string root occurs more than once on a line only the first match will be replaced.

To tell sed to do a global substitution, add the letter **g** to the end of the command as follows:

```
$ cat /etc/passwd | sed 's/root/amrood/g'
amrood:x:0:0:amrood user:/amrood:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

```
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
.........................
```

Substitution Flags:

There are a number of other useful flags that can be passed in addition to the g flag, and you can specify more than one at a time.

Flag	Description
g	Replace all matches, not just the first match.
NUMBER	Replace only NUMBERth match.
p	If substitution was made, print pattern space.
w FILENAME	If substitution was made, write result to FILENAME.
I or i	Match in a case-insensitive manner.
M or m	In addition to the normal behavior of the special regular expression characters ^ and $, this flag causes ^ to match the empty string after a newline and $ to match the empty string before a newline.

Using an Alternative String Separator:

You may find yourself having to do a substitution on a string that includes the forward slash character. In this case, you can specify a different separator by providing the designated character after the s.

```
$ cat /etc/passwd | sed 's:/root:/amrood:g'
amrood:x:0:0:amrood user:/amrood:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

In the above example we have used **:** as delimeter instead of slash / because we were trying to search /root instead of simple root.

Replacing with Empty Space:

Use an empty substitution string to delete the root string from the /etc/passwd file entirely:

```
$ cat /etc/passwd | sed 's/root//g'
:x:0:0::/:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

Address Substitution:

If you want to substitute the string sh with the string quiet only on line 10, you can specify it as follows:

```
$ cat /etc/passwd | sed '10s/sh/quiet/g'
root:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
```

```
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/quiet
```

Similarly, to do an address range substitution, you could do something like the following:

```
$ cat /etc/passwd | sed '1,5s/sh/quiet/g'
root:x:0:0:root user:/root:/bin/quiet
daemon:x:1:1:daemon:/usr/sbin:/bin/quiet
bin:x:2:2:bin:/bin:/bin/quiet
sys:x:3:3:sys:/dev:/bin/quiet
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

As you can see from the output, the first five lines had the string sh changed to quiet, but the rest of the lines were left untouched.

The Matching Command:

You would use **p** option along with **-n** option to print all the matching lines as follows:

```
$ cat testing | sed -n '/root/p'
root:x:0:0:root user:/root:/bin/sh
[root@ip-72-167-112-17 amrood]# vi testing
root:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Using Regular Expression:

While matching pattern, you can use regular expression which provides more flexibility.

Check following example which matches all the lines starting with *daemon* and then deleting them:

```
$ cat testing | sed '/^daemon/d'
root:x:0:0:root user:/root:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

```
backup:x:34:34:backup:/var/backups:/bin/sh
```

Following is the example which would delete all the lines ending with **sh**:

```
$ cat testing | sed '/sh$/d'
sync:x:4:65534:sync:/bin:/bin/sync
```

The following table lists four special characters that are very useful in regular expressions.

Character	Description
^	Matches the beginning of lines.
$	Matches the end of lines.
.	Matches any single character.
*	Matches zero or more occurrences of the previous character
[chars]	Matches any one of the characters given in chars, where chars is a sequence of characters. You can use the - character to indicate a range of characters.

Matching Characters:

Look at a few more expressions to demonstrate the use of the metacharacters. For example, the following pattern:

Expression	Description
/a.c/	Matches lines that contain strings such as a+c, a-c, abc, match, and a3c, whereas the pattern
/a*c/	Matches the same strings along with strings such as ace, yacc, and arctic.
/[tT]he/	Matches the string The and the:
/^$/	Matches Blank lines
/^.*$/	Matches an entire line whatever it is.
/ */	Matches one or more spaces
/^$/	Matches Blank lines

Following table shows some frequently used sets of characters:

Set	Description
[a-z]	Matches a single lowercase letter
[A-Z]	Matches a single uppercase letter
[a-zA-Z]	Matches a single letter

[0-9]	Matches a single number
[a-zA-Z0-9]	Matches a single letter or number

Character Class Keywords:

Some special keywords are commonly available to regexps, especially GNU utilities that employ regexps. These are very useful for sed regular expressions as they simplify things and enhance readability.

For example, the characters a through z as well as the characters A through Z constitute one such class of characters that has the keyword [[:alpha:]]

Using the alphabet character class keyword, this command prints only those lines in the /etc/syslog.conf file that start with a letter of the alphabet:

```
$ cat /etc/syslog.conf | sed -n '/^[[:alpha:]]/p'
authpriv.*                          /var/log/secure
mail.*                              -/var/log/maillog
cron.*                              /var/log/cron
uucp,news.crit                      /var/log/spooler
local7.*                            /var/log/boot.log
```

The following table is a complete list of the available character class keywords in GNU sed.

Character Class	Description
[[:alnum:]]	Alphanumeric [a-z A-Z 0-9]
[[:alpha:]]	Alphabetic [a-z A-Z]
[[:blank:]]	Blank characters (spaces or tabs)
[[:cntrl:]]	Control characters
[[:digit:]]	Numbers [0-9]
[[:graph:]]	Any visible characters (excludes whitespace)
[[:lower:]]	Lowercase letters [a-z]
[[:print:]]	Printable characters (noncontrol characters)
[[:punct:]]	Punctuation characters
[[:space:]]	Whitespace
[[:upper:]]	Uppercase letters [A-Z]
[[:xdigit:]]	Hex digits [0-9 a-f A-F]

Aampersand Referencing:

The sed metacharacter & represents the contents of the pattern that was matched. For instance, say you have a file called phone.txt full of phone numbers, such as the following:

```
5555551212
5555551213
5555551214
6665551215
6665551216
7775551217
```

You want to make the area code (the first three digits) surrounded by parentheses for easier reading. To do this, you can use the ampersand replacement character, like so:

```
$ sed -e 's/^[[:digit:]][[:digit:]][[:digit:]]/(&)/g' phone.txt
(555)5551212
(555)5551213
(555)5551214
(666)5551215
(666)5551216
(777)5551217
```

Here in pattern part you are matching first 3 digits and then using & you are replacing those 3 digits with surrounding parentheses.

Using Multiple sed Commands:

You can use multiple sed commands in a single sed command as follows:

```
$ sed -e 'command1' -e 'command2' ... -e 'commandN' files
```

Here command1 through commandN are sed commands of the type discussed previously. These commands are applied to each of the lines in the list of files given by files.

Using the same mechanism, we can write above phone number example as follows:

```
$ sed -e 's/^[[:digit:]]\{3\}/(&)/g'  \
                      -e 's/)[[:digit:]]\{3\}/&-/g' phone.txt
(555)555-1212
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
(777)555-1217
```

Note: In the above example, instead of repeating the character class keyword [[:digit:]] three times, you replaced it with \{3\}, which means to match the preceding regular expression three times. Here I used \ to give line break you should remove this before running this command.

Back References:

The ampersand metacharacter is useful, but even more useful is the ability to define specific regions in a regular expressions so you can reference them in your replacement strings. By defining specific parts of a regular expression, you can then refer back to those parts with a special reference character.

To do back references, you have to first define a region and then refer back to that region. To define a region you insert backslashed parentheses around each region of interest. The first region that you surround with backslashes is then referenced by \1, the second region by \2, and so on.

Assuming phone.txt has the following text:

```
(555)555-1212
```

```
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
(777)555-1217
```

Now try the following command:

```
$ cat phone.txt | sed 's/\(.*\)\(.*-\)\(.*$\)/Area \
                      code: \1 Second: \2 Third: \3/'
Area code: (555) Second: 555- Third: 1212
Area code: (555) Second: 555- Third: 1213
Area code: (555) Second: 555- Third: 1214
Area code: (666) Second: 555- Third: 1215
Area code: (666) Second: 555- Third: 1216
Area code: (777) Second: 555- Third: 1217
```

Note: In the above example each regular expression inside the parenthesis would be back referenced by \1, \2 and so on. Here I used \ to give line break you should remove this before running this command.

CHAPTER

25

Unix – File System Basics

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Your hard drive can have various partitions which usually contains only one file system, such as one file system housing the / file system or another containing the /home file system.

One file system per partition allows for the logical maintenance and management of differing file systems.

Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, floppy drives, and so forth.

Directory Structure:

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 0 and 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 any other.

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:

Directory	Description
/	This is the root directory which should contain only the directories needed at the top level of the file structure.
/bin	This is where the executable files are located. They are available to all user.

/dev	These are device drivers.
/etc	Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages.
/lib	Contains shared library files and sometimes other kernel-related files.
/boot	Contains files for booting the system.
/home	Contains the home directory for users and other accounts.
/mnt	Used to mount other temporary file systems, such as cdrom and floppy for the CD-ROM drive and floppy diskette drive, respectively
/proc	Contains all processes marked as a file by process number or other information that is dynamic to the system.
/tmp	Holds temporary files used between system boots
/usr	Used for miscellaneous purposes, or can be used by many users. Includes administrative commands, shared files, library files, and others
/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
/sbin	Contains binary (executable) files, usually for system administration. For example*fdisk* and *ifconfig* utlities.
/kernel	Contains kernel files

Navigating the File System:

Now that you understand the basics of the file system, you can begin navigating to the files you need. The following are commands you'll use to navigate the system:

Command	Description
cat filename	Displays a filename.
cd dirname	Moves you to the directory identified.
cp file1 file2	Copies one file/directory to specified location.
file filename	Identifies the file type (binary, text, etc).
find filename dir	Finds a file/directory.
head filename	Shows the beginning of a file.
less filename	Browses through a file from end or beginning.
ls dirname	Shows the contents of the directory specified.
mkdir dirname	Creates the specified directory.
more filename	Browses through a file from beginning to end.
mv file1 file2	Moves the location of or renames a file/directory.
pwd	Shows the current directory the user is in.
rm filename	Removes a file.
rmdir dirname	Removes a directory.

tail filename	Shows the end of a file.
touch filename	Creates a blank file or modifies an existing file.s attributes.
whereis filename	Shows the location of a file.
which filename	Shows the location of a file if it is in your PATH.

You can use Manpage Help to check complete syntax for each command mentioned here.

The df Command:

The first way to manage your partition space is with the df (disk free) command. The command df -k (disk free) displays the disk space usage in kilobytes, as shown below:

```
$df -k
Filesystem      1K-blocks      Used   Available Use% Mounted on
/dev/vzfs        10485760   7836644     2649116  75% /
/devices                0         0           0   0% /devices
$
```

Some of the directories, such as /devices, shows 0 in the kbytes, used, and avail columns as well as 0% for capacity. These are special (or virtual) file systems, and although they reside on the disk under /, by themselves they do not take up disk space.

The df -k output is generally the same on all Unix systems. Here's what it usually includes:

Column	Description
Filesystem	The physical file system name.
Kbytes	Total kilobytes of space available on the storage medium.
Used	Total kilobytes of space used (by files).
Avail	Total kilobytes available for use.
Capacity	Percentage of total space used by files.
Mounted on	What the file system is mounted on.

You can use the -h (human readable) option to display the output in a format that shows the size in easier-to-understand notation.

The du Command:

The du (disk usage) command enables you to specify directories to show disk space usage on a particular directory.

This command is helpful if you want to determine how much space a particular directory is taking. Following command would display number of blocks consumed by each directory. A single block may take either 512 Bytes or 1 Kilo Byte depending on your system.

```
$du /etc
10      /etc/cron.d
126     /etc/default
6       /etc/dfs
...
$
```

The -h option makes the output easier to comprehend:

```
$du -h /etc
5k    /etc/cron.d
63k   /etc/default
3k    /etc/dfs
...
$
```

Mounting the File System:

A file system must be mounted in order to be usable by the system. To see what is currently mounted (available for use) on your system, use this command:

```
$ mount
/dev/vzfs on / type reiserfs (rw,usrquota,grpquota)
proc on /proc type proc (rw,nodiratime)
devpts on /dev/pts type devpts (rw)
$
```

The /mnt directory, by Unix convention, is where temporary mounts (such as CD-ROM drives, remote network drives, and floppy drives) are located. If you need to mount a file system, you can use the mount command with the following syntax:

```
mount -t file_system_type device_to_mount directory_to_mount_to
```

For example, if you want to mount a CD-ROM to the directory /mnt/cdrom, for example, you can type:

```
$ mount -t iso9660 /dev/cdrom /mnt/cdrom
```

This assumes that your CD-ROM device is called /dev/cdrom and that you want to mount it to /mnt/cdrom. Refer to the mount man page for more specific information or type mount -h at the command line for help information.

After mounting, you can use the cd command to navigate the newly available file system through the mountpoint you just made.

Unmounting the File System:

To unmount (remove) the file system from your system, use the **umount** command by identifying the mountpoint or device

For example, to unmount cdrom, use the following command:

```
$ umount /dev/cdrom
```

The mount command enables you to access your file systems, but on most modern Unix systems, the automount function makes this process invisible to the user and requires no intervention.

User and Group Quotas:

User and group quotas provide the mechanisms by which the amount of space used by a single user or all users within a specific group can be limited to a value defined by the administrator.

Quotas operate around two limits that allow the user to take some action if the amount of space or number of disk blocks start to exceed the administrator defined limits:

- **Soft Limit:** If the user exceeds the limit defined, there is a grace period that allows the user to free up some space.
- **Hard Limit:** When the hard limit is reached, regardless of the grace period, no further files or blocks can be allocated.

There are a number of commands to administer quotas:

Command	Description
quota	Displays disk usage and limits for a user of group.
edquota	This is a quota editor. Users or Groups quota can be edited using this command.
quotacheck	Scan a filesystem for disk usage, create, check and repair quota files
setquota	This is also a command line quota editor.
quotaon	This announces to the system that disk quotas should be enabled on one or more filesystems.
quotaoff	This announces to the system that disk quotas should be disabled off one or more filesystems.
repquota	This prints a summary of the disc usage and quotas for the specified file systems

You can use Manpage Help to check complete syntax for each command mentioned here.

CHAPTER

26

Unix – User Administration

There are three types of accounts on a Unix system:

1. **Root account:** This is also called superuser and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.
2. **System accounts:** System accounts are those needed for the operation of system-specific components for example mail accounts and the sshd accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system.
3. **User accounts:** User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Unix supports a concept of *Group Account* which logically groups a number of accounts. Every account would be a part of any group account. Unix groups plays important role in handling file permissions and process management.

Managing Users and Groups:

There are three main user administration files:

1. **/etc/passwd:** Keeps user account and password information. This file holds the majority of information about accounts on the Unix system.
2. **/etc/shadow:** Holds the encrypted password of the corresponding account. Not all the system support this file.
3. **/etc/group:** This file contains the group information for each account.
4. **/etc/gshadow:** This file contains secure group account information.

Check all the above files using **cat** command.

Following are commands available on the majority of Unix systems to create and manage accounts and groups:

Command	Description
useradd	Adds accounts to the system.
usermod	Modifies account attributes.
userdel	Deletes accounts from the system.
groupadd	Adds groups to the system.

groupmod	Modifies group attributes.
groupdel	Removes groups from the system.

You can use Manpage Help to check complete syntax for each command mentioned here.

Create a Group

You would need to create groups before creating any account otherwise you would have to use existing groups at your system. You would have all the groups listed in */etc/groups* file.

All the default groups would be system account specific groups and it is not recommended to use them for ordinary accounts. So following is the syntax to create a new group account:

```
groupadd [-g gid [-o]] [-r] [-f] groupname
```

Here is the detail of the parameters:

Option	Description
-g GID	The numerical value of the group's ID.
-o	This option permits to add group with non-unique GID
-r	This flag instructs groupadd to add a system account
-f	This option causes to just exit with success status if the specified group already exists. With -g, if specified GID already exists, other (unique) GID is chosen
Groupname	Actaul group name to be created.

If you do not specify any parameter then system would use default values.

Following example would create *developers* group with default values, which is very much acceptable for most of the administrators.

```
$ groupadd developers
```

Modify a Group:

To modify a group, use the **groupmod** syntax:

```
$ groupmod -n new_modified_group_name old_group_name
```

To change the developers_2 group name to developer, type:

```
$ groupmod -n developer developer_2
```

Here is how you would change the financial GID to 545:

```
$ groupmod -g 545 developer
```

Delete a Group:

To delete an existing group, all you need are the groupdel command and the group name. To delete the financial group, the command is:

```
$ groupdel developer
```

This removes only the group, not any files associated with that group. The files are still accessible by their owners.

Create an Account

Let us see how to create a new account on your Unix system. Following is the syntax to create a user's account:

```
useradd -d homedir -g groupname -m -s shell -u userid accountname
```

Here is the detail of the parameters:

Option	Description
-d homedir	Specifies home directory for the account.
-g groupname	Specifies a group account for this account.
-m	Creates the home directory if it doesn't exist.
-s shell	Specifies the default shell for this account.
-u userid	You can specify a user id for this account.
Accountname	Actual account name to be created

If you do not specify any parameter then system would use default values. The useradd command modifies the /etc/passwd, /etc/shadow, and /etc/group files and creates a home directory.

Following is the example which would create an account *mcmohd* setting its home directory to*/home/mcmohd* and group as *developers*. This user would have Korn Shell assigned to it.

```
$ useradd -d /home/mcmohd -g developers -s /bin/ksh mcmohd
```

Before issuing above command, make sure you already have *developers* group created using*groupadd* command.

Once an account is created you can set its password using the **passwd** command as follows:

```
$ passwd mcmohd20
Changing password for user mcmohd20.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

When you type *passwd accountname*, it gives you option to change the password provided you are super user otherwise you would be able to change just your password using the same command but without specifying your account name.

Modify an Account:

The **usermod** command enables you to make changes to an existing account from the command line. It uses the same arguments as the useradd command, plus the -l argument, which allows you to change the account name. For example, to change the account name *mcmohd* to *mcmohd20* and to change home directory accordingly, you would need to issue following command:

```
$ usermod -d /home/mcmohd20 -m -l mcmohd mcmohd20
```

Delete an Account:

The **userdel** command can be used to delete an existing user. This is a very dangerous command if not used with caution.

There is only one argument or option available for the command: .r, for removing the account's home directory and mail file.

For example, to remove account *mcmohd20*, you would need to issue following command:

```
$ userdel -r mcmohd20
```

If you want to keep her home directory for backup purposes, omit the -r option. You can remove the home directory as needed at a later time.

CHAPTER

27

Unix – System Performance

The purpose of this tutorial is to introduce the performance analyst to some of the free tools available to monitor and manage performance on UNIX systems, and to provide a guideline on how to diagnose and fix performance problems in Unix environment.

UNIX has following major resource types that need to be monitored and tuned:

- **CPU**
- **Memory**
- **Disk space**
- **Communications lines**
- **I/O Time**
- **Network Time**
- **Applications programs**

Performance Components:

There are following major five component where total system time goes:

Component	Description
User state CPU	The actual amount of time the CPU spends running the users program in the user state. It includes time spent executing library calls, but does not include time spent in the kernel on its behalf.
System state CPU	This is the amount of time the CPU spends in the system state on behalf of this program. All I/O routines require kernel services. The programmer can affect this value by the use of blocking for I/O transfers.
I/O Time and Network Time	These are the amount of time spent moving data and servicing I/O requests
Virtual Memory Performance	This includes context switching and swapping.
Application Program	Time spent running other programs - when the system is not servicing this application because another application currently has the CPU.

Performance Tools:

Unix provides following important tools to measure and fine tune Unix system performance:

Command	Description
nice/renice	Run a program with modified scheduling priority
Netstat	Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships
Time	Time a simple command or give resource usage
Uptime	System Load Average
Ps	Report a snapshot of the current processes.
Vmstat	Report virtual memory statistics
Gprof	Display call graph profile data
Prof	Process Profiling
Top	Display system tasks

You can use Manpage Help to check complete syntax for each command mentioned here.

CHAPTER

28

Unix – System Logging

Unix systems have a very flexible and powerful logging system, which enables you to record almost anything you can imagine and then manipulate the logs to retrieve the information you require.

Many versions of UNIX provide a general-purpose logging facility called *syslog*. Individual programs that need to have information logged send the information to syslog.
Unix *syslog* is a host-configurable, uniform system logging facility. The system uses a centralized system logging process that runs the program **/etc/syslogd** or **/etc/syslog**.
The operation of the system logger is quite straightforward. Programs send their log entries to *syslogd*, which consults the configuration file /etc/syslogd.conf or /etc/syslog and, when a match is found, writes the log message to the desired log file.

There are four basic syslog terms that you should understand:

Term	Description
Facility	The identifier used to describe the application or process that submitted the log message. Examples are mail, kernel, and ftp.
Priority	An indicator of the importance of the message. Levels are defined within syslog as guidelines, from debugging information to critical events.
Selector	A combination of one or more facilities and levels. When an incoming event matches a selector, an action is performed.
Action	What happens to an incoming message that matches a selector. Actions can write the message to a log file, echo the message to a console or other device, write the message to a logged in user, or send the message along to another syslog server.

Syslog Facilities:

Here are the available facilities for the selector. Not all facilities are present on all versions of UNIX.

Facility	Description
auth	Activity related to requesting name and password (getty, su, login)
authpriv	Same as auth but logged to a file that can only be read by selected users

console	Used to capture messages that would generally be directed to the system console
cron	Messages from the cron system scheduler
daemon	System daemon catch-all
ftp	Messages relating to the ftp daemon
kern	Kernel messages
local0.local7	Local facilities defined per site
lpr	Messages from the line printing system
mail	Messages relating to the mail system
mark	Pseudo event used to generate timestamps in log files
news	Messages relating to network news protocol (nntp)
ntp	Messages relating to network time protocol
user	Regular user processes
uucp	UUCP subsystem

Syslog Priorities:

The syslog priorities are summarized in the following table:

Priority	Description
Emerg	Emergency condition, such as an imminent system crash, usually broadcast to all users
Alert	Condition that should be corrected immediately, such as a corrupted system database
Crit	Critical condition, such as a hardware error
Err	Ordinary error
Warning	Warning
Notice	Condition that is not an error, but possibly should be handled in a special way
Info	Informational message
Debug	Messages that are used when debugging programs
None	Pseudo level used to specify not to log messages.

The combination of facilities and levels enables you to be discerning about what is logged and where that information goes.

As each program sends its messages dutifully to the system logger, the logger makes decisions on what to keep track of and what to discard based on the levels defined in the selector.

When you specify a level, the system will keep track of everything at that level and higher.

The /etc/syslog.conf file:

The /etc/syslog.conf file controls where messages are logged. A typical syslog.conf file might look like this:

```
*.err;kern.debug;auth.notice /dev/console
daemon,auth.notice           /var/log/messages
lpr.info                     /var/log/lpr.log
mail.*                       /var/log/mail.log
ftp.*                        /var/log/ftp.log
auth.*                       @prep.ai.mit.edu
auth.*                       root,amrood
netinfo.err                  /var/log/netinfo.log
install.*                    /var/log/install.log
*.emerg                      *
*.alert                      |program_name
mark.*                       /dev/console
```

Each line of the file contains two parts:

- A message selector that specifies which kind of messages to log. For example, all error messages or all debugging messages from the kernel.

- An action field that says what should be done with the message. For example, put it in a file or send the message to a user's terminal.

Following are the notable points for the above configuration:

- Message selectors have two parts: a facility and a priority. For example, *kern.debug* selects all debug messages (the priority) generated by the kernel (the facility).

- Message selectetor *kern.debug* selects all priorities that are greater than debug.

- An asterisk in place of either the facility or the priority indicates "all." For example, *.debug means all debug messages, while kern.* means all messages generated by the kernel.

- You can also use commas to specify multiple facilities. Two or more selectors can be grouped together by using a semicolon.

Logging Actions:

The action field specifies one of five actions:

1. Log message to a file or a device. For example, /var/log/lpr.log or /dev/console.
2. Send a message to a user. You can specify multiple usernames by separating them with commas (e.g., root, amrood).
3. Send a message to all users. In this case, the action field consists of an asterisk (e.g., *).
4. Pipe the message to a program. In this case, the program is specified after the UNIX pipe symbol (|).
5. Send the message to the syslog on another host. In this case, the action field consists of a hostname, preceded by an at sign (e.g., @tutorialspoint.com)

The logger Command:

UNIX provides the **logger** command, which is an extremely useful command to deal with system logging. The **logger** command sends logging messages to the syslogd daemon, and consequently provokes system logging.
This means we can check from the command line at any time the **syslogd** daemon and its configuration. The logger command provides a method for adding one-line entries to the system log file from the command line.

The format of the command is:

```
logger [-i] [-f file] [-p priority] [-t tag] [message]...
```

Here is the detail of the parameters:

Option	Description
-f filename	Use the contents of file filename as the message to log.
-i	Log the process ID of the logger process with each line.
-p priority	Enter the message with the specified priority (specified selector entry); the message priority can be specified numerically, or as a facility.priority pair. The default priority is user.notice.
-t tag	Mark each line added to the log with the specified tag.
message	The string arguments whose contents are concatenated together in the specified order, separated by the space

You can use Manpage Help to check complete syntax for this command.

Log Rotation:

Log files have the propensity to grow very fast and consume large amounts of disk space. To enable log rotations, most distributions use tools such as *newsyslog* or *logrotate*.

These tools should be called on a frequent time interval using the cron daemon. Check the man pages for *newsyslog* or *logrotate* for more details.

Important Log Locations

All the system applications create their log files in */var/log* and its sub-directories. Here are few important applications and their coressponding log directories:

Application	Directory
Httpd	/var/log/httpd
Samba	/var/log/samba
Cron	/var/log/
Mail	/var/log/
Mysql	/var/log/

CHAPTER

29

Unix – Signals and Traps

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

The following are some of the more common signals you might encounter and want to use in your programs:

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).

List of Signals:

There is an easy way to list down all the signals supported by your system. Just issue **kill -l** command and it would display all the supported signals:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
```

```
39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12
47) SIGRTMIN+13  48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

The actual list of signals varies between Solaris, HP-UX, and Linux.

Default Actions:

Every signal has a default action associated with it. The default action for a signal is the action that a script or program performs when it receives a signal.

Some of the possible default actions are:

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called core containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

Sending Signals:

There are several methods of delivering signals to a program or script. One of the most common is for a user to type CONTROL-C or the INTERRUPT key while a script is executing.

When you press the *Ctrl+C* key a SIGINT is sent to the script and as per defined default action script terminates.

The other common method for delivering signals is to use the kill command whose syntax is as follows:

```
$ kill -signal pid
```

Here **signal** is either the number or name of the signal to deliver and **pid** is the process ID that the signal should be sent to. For Example:

```
$ kill -1 1001
```

Sends the HUP or hang-up signal to the program that is running with process ID 1001. To send a kill signal to the same process use the following command:

```
$ kill -9 1001
```

This would kill the process running with process ID 1001.

Trapping Signals:

When you press the *Ctrl+C* or Break key at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returned. This may not always be desirable. For instance, you may end up leaving a bunch of temporary files that won't get cleaned up.

Trapping these signals is quite easy, and the trap command has the following syntax:

```
$ trap commands signals
```

Here *command* can be any valid Unix command, or even a user-defined function, and signal can be a list of any number of signals you want to trap.

There are three common uses for trap in shell scripts:

1. Clean up temporary files

2. Ignore signals

Cleaning Up Temporary Files:

As an example of the trap command, the following shows how you can remove some files and then exit if someone tries to abort the program from the terminal:

```
$ trap "rm -f $WORKDIR/work1$$ $WORKDIR/dataout$$; exit" 2
```

From the point in the shell program that this trap is executed, the two files *work1$$* and *dataout$$* will be automatically removed if signal number 2 is received by the program.

So if the user interrupts execution of the program after this trap is executed, you can be assured that these two files will be cleaned up. The **exit** command that follows the **rm** is necessary because without it execution would continue in the program at the point that it left off when the signal was received.

Signal number 1 is generated for hangup: Either someone intentionally hangs up the line or the line gets accidentally disconnected.

You can modify the preceding trap to also remove the two specified files in this case by adding signal number 1 to the list of signals:

```
$ trap "rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit" 1 2
```

Now these files will be removed if the line gets hung up or if the *Ctrl+C* key gets pressed.

The commands specified to trap must be enclosed in quotes if they contain more than one command. Also note that the shell scans the command line at the time that the trap command gets executed and also again when one of the listed signals is received.

So in the preceding example, the value of WORKDIR and $$ will be substituted at the time that the trap command is executed. If you wanted this substitution to occur at the time that either signal 1 or 2 was received you can put the commands inside single quotes:

```
$ trap 'rm $WORKDIR/work1$$ $WORKDIR/dataout$$; exit' 1 2
```

Ignoring Signals:

If the command listed for trap is null, the specified signal will be ignored when received. For example, the command:

```
$ trap '' 2
```

Specifies that the interrupt signal is to be ignored. You might want to ignore certain signals when performing some operation that you don't want interrupted. You can specify multiple signals to be ignored as follows:

```
$ trap '' 1 2 3 15
```

Note that the first argument must be specified for a signal to be ignored and is not equivalent to writing the following, which has a separate meaning of its own:

```
$ trap  2
```

If you ignore a signal, all subshells also ignore that signal. However, if you specify an action to be taken on receipt of a signal, all subshells will still take the default action on receipt of that signal.

Resetting Traps:

After you've changed the default action to be taken on receipt of a signal, you can change it back again with trap if you simply omit the first argument; so

```
$ trap 1 2
```

resets the action to be taken on receipt of signals 1 or 2 back to the default.

CHAPTER

30

Unix – Useful Commands

This quick guide lists commands, including a syntax and brief description. For more detail, use:

```
$man command
```

Files and Directories:

These commands allow you to create directories and handle files.

Command	Description
Cat	Display File Contents
Cd	Changes Directory to dirname
Chgrp	change file group
Chmod	Changing Permissions
Cp	Copy source file into destination
File	Determine file type
Find	Find files
Grep	Search files for regular expressions.
Head	Display first few lines of a file
Ln	Create softlink on oldname
Ls	Display information about file type.
Mkdir	Create a new directory dirname
More	Display data in paginated form.
Mv	Move (Rename) a oldname to newname.

Pwd	Print current working directory.
Rm	Remove (Delete) filename
Rmdir	Delete an existing directory provided it is empty.
Tail	Prints last few lines in a file.
Touch	Update access and modification time of a file.

Manipulating data:

The contents of files can be compared and altered with the following commands.

Command	Description
Awk	Pattern scanning and processing language
Cmp	Compare the contents of two files
Comm.	Compare sorted data
Cut	Cut out selected fields of each line of a file
Diff	Differential file comparator
Expand	Expand tabs to spaces
Join	Join files on some common field
Perl	Data manipulation language
Sed	Stream text editor
Sort	Sort file data
Split	Split file into smaller files
Tr	Translate characters
Uniq	Report repeated lines in a file
Wc	Count words, lines, and characters
Vi	Opens vi text editor
Vim	Opens vim text editor
Fmt	Simple text formatter
Spell	Check text for spelling error
Ispell	Check text for spelling error

Ispell	Check text for spelling error
Emacs	GNU project Emacs
ex, edit	Line editor
Emacs	GNU project Emacs
Emacs	GNU project Emacs

Compressed Files:

Files may be compressed to save space. Compressed files can be created and examined:

Command	Description
compress	Compress files
Gunzip	Uncompress gzipped files
Gzip	GNU alternative compression method
uncompress	Uncompress files
Unzip	List, test and extract compressed files in a ZIP archive
Zcat	Cat a compressed file
Zcmp	Compare compressed files
Zdiff	Compare compressed files
Zmore	File perusal filter for crt viewing of compressed text

Getting Information:

Various Unix manuals and documentation are available on-line. The following Shell commands give information:

Command	Description
apropos	Locate commands by keyword lookup
Info	Displays command information pages online
Man	Displays manual pages online
Whatis	Search the whatis database for complete words.
Yelp	GNOME help viewer

Network Communication:

These following commands are used to send and receive files from a local UNIX hosts to the remote host around the world.

Command	Description
ftp	File transfer program
Rcp	Remote file copy
rlogin	Remote login to a UNIX host
Rsh	Remote shell
Tftp	Trivial file transfer program
telnet	Make terminal connection to another host
Ssh	Secure shell terminal or command connection
Scp	Secure shell remote file copy
Sftp	secure shell file transfer program

Some of these commands may be restricted at your computer for security reasons.

Messages between Users:

The UNIX systems support on-screen messages to other users and world-wide electronic mail:

Command	Description
evolution	GUI mail handling tool on Linux
Mail	Simple send or read mail program
Mesg	Permit or deny messages
Parcel	Send files to another user
Pine	Vdu-based mail utility
Talk	Talk to another user
Write	Write message to another user

Programming Utilities:

The following programming tools and languages are available based on what you have installed on your Unix.

Command	Description
Dbx	Sun debugger
Gdb	GNU debugger
Make	Maintain program groups and compile programs.
Nm	Print program's name list
Size	Print program's sizes
Strip	Remove symbol table and relocation bits
Cb	C program beautifier
Cc	ANSI C compiler for Suns SPARC systems
Ctrace	C program debugger
Gcc	GNU ANSI C Compiler
Indent	Indent and format C program source
Bc	Interactive arithmetic language processor
Gcl	GNU Common Lisp
Perl	General purpose language
Php	Web page embedded language
Py	Python language interpreter
Asp	Web page embedded language
CC	C++ compiler for Suns SPARC systems
g++	GNU C++ Compiler
Javac	JAVA compiler
appletvieweir	JAVA applet viewer
netbeans	Java integrated development environment on Linux
Sqlplus	Run the Oracle SQL interpreter
Sqlldr	Run the Oracle SQL data loader
Mysql	Run the mysql SQL interpreter

Misc Commands:

These commands list or alter information about the system:

Command	Description
Chfn	Change your finger information
Chgrp	Change the group ownership of a file
Chown	Change owner
Date	Print the date
determin	Automatically find terminal type
Du	Print amount of disk usage
Echo	Echo arguments to the standard options
Exit	Quit the system
Finger	Print information about logged-in users
groupadd	Create a user group
Groups	Show group memberships
homequota	Show quota and file usage
Iostat	Report I/O statistics
Kill	Send a signal to a process
Last	Show last logins of users
Logout	log off UNIX
Lun	List user names or login ID
Netstat	Show network status
Passwd	Change user password
Passwd	Change your login password
printenv	Display value of a shell variable
Ps	Display the status of current processes
Ps	Print process status statistics
quota -v	Display disk usage and limits

Reset	Reset terminal mode
Script	Keep script of terminal session
Script	Save the output of a command or process
Setenv	Set environment variables
Sty	Set terminal options
Time	Time a command
Top	Display all system processes
Tset	Set terminal mode
Tty	Print current terminal name
Umask	Show the permissions that are given to view files by default
Uname	Display name of the current system
Uptime	Get the system up time
useradd	Create a user account
Users	Print names of logged in users
Vmstat	Report virtual memory statistics
W	Show what logged in users are doing
Who	List logged in users

CHAPTER

31

Unix – Builtin Functions

The most of the part of this tutorial covered Bourne Shell but this page list down all the mathematical builti-in functions available in Korn Shell.

The Korn shell provides access to the standard set of mathematical functions. They are called using C function call syntax.

Function	Description
Abs	Absolute value
Log	Natural logarithm
Acos	Arc cosine
Sin	Sine
Asin	Arc sine
Sinh	Hyperbolic sine
Cos	Cosine
Sqrt	Square root
Cosh	Hyperbolic cosine
Tan	Tangent
Exp	Exponential function
Tanh	Hyperbolic tangent
Int	Integer part of floating-point number