http://www.tutorialspoint.com/unix/unix-file-management.htm

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 the 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.

Execute Unix Shell Programs

If you are willing to learn the Unix/Linux basic commands and shell script but you do not have a setup for the same, then do not worry. The [Coding Ground](http://www.tutorialspoint.com/codingground.htm) is available on a high end dedicated server giving you real programming experience with a comfort of single click execution. Yes! it is absolutely free and its online.

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, task 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

* Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
* 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.
* Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.
* 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 −

* To start, type **passwd** at command prompt as shown below.
* Enter your old password the one you're currently using.
* Type in your new password. Always keep your password complex enough so that no body can guess it. But make sure, you remember it.
* 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

* 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.

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 −

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

* First Column: represents file type and permission given on the file. Below is the description of all type of files.
* Second Column: represents the number of memory blocks taken by the file or directory.
* Third Column: represents owner of the file. This is the Unix user who created this file.
* Fourth Column: represents group of the owner. Every Unix user would have an associated group.
* Fifth Column: represents file size in bytes.
* Sixth Column: represents date and time when this file was created or modified last time.
* 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.

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 **filename** 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 -b filename

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 −

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

* **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.
* **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
* **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.

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 remove 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

$

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-x**r--** represents that owner 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-x**r--** 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-x**r--** 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=rx 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=rx 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 −

* **chown** − The chown command stands for "change owner" and is used to change the owner of a file.
* **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

$

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 environment. This is usually a two step process that involves the shell reading the following files −

* /etc/profile
* profile

The process is as follows −

* The shell checks to see whether the file **/etc/profile** exists.
* If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
* 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.
* 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

$

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-x**r--** represents that owner 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-x**r--** 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-x**r--** 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=rx 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=rx 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 −

* **chown** − The chown command stands for "change owner" and is used to change the owner of a file.
* **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

$

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

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.

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 fields 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.

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 remote 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.

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 **V**i **IM**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 −

* **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.
* **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+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+I | 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 Paste 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.

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:

* The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
* 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 –

**$ ls -lrt**

total 12

-rw-r--r-- 1 sm70471 gpa 74 Oct 25 23:58 test.sh

**-rw-r--r-- 1 sm70471 gpa 62 Oct 25 23:59 test1.sh**

-rw-r--r-- 1 sm70471 gpa 55 Oct 26 00:23 test

**$ chmod +x test1.sh**

**$ ls -lrt**

total 12

-rw-r--r-- 1 sm70471 gpa 74 Oct 25 23:58 test.sh

**-rwxr-xr-x 1 sm70471 gpa 62 Oct 25 23:59 test1.sh**

-rw-r--r-- 1 sm70471 gpa 55 Oct 26 00:23 test

**$ ./test1.sh -- To execute the script use ./**

what is your name

ambika

your name is ambika

$./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

$

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.
* Previous tutorial 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 "Second 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
* Second 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
* Second Parameter : Ali
* Quoted Values: Zara Ali
* Quoted Values: Zara Ali
* Total Number of Parameters : 2
* $echo $?
* 0
* $

**$ cat scrt1.sh**

**#!/bin/sh**

**echo "FILENAME : $0"**

**echo " FIRST PARAMETER : $1"**

**echo "SECOND PARAMETER : $2"**

**echo "QUOTED VALUES : $@"**

**echo "QUOTED VALUES : $\*"**

**echo "Number of Arguments : $#"**

**$ ./scrt1.sh sree ambika mattaparthi**

**FILENAME : ./scrt1.sh**

**FIRST PARAMETER : sree**

**SECOND PARAMETER : ambika**

**QUOTED VALUES : sree ambika mattaparthi**

**QUOTED VALUES : sree ambika mattaparthi**

**Number of Arguments : 3**

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

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 −

[Show Examples](https://www.tutorialspoint.com/unix/unix-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| 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.

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 −

[Show Examples](https://www.tutorialspoint.com/unix/unix-relational-operators.htm)

|  |  |  |
| --- | --- | --- |
| 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.

Boolean Operators

There are following boolean operators supported by Bourne Shell.

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

[Show Examples](https://www.tutorialspoint.com/unix/unix-boolean-operators.htm)

|  |  |  |
| --- | --- | --- |
| 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. |

String Operators

There are following string operators supported by Bourne Shell.

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

[Show Examples](https://www.tutorialspoint.com/unix/unix-string-operators.htm)

|  |  |  |
| --- | --- | --- |
| 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. | [ -n $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. |

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 −

[Show Examples](https://www.tutorialspoint.com/unix/unix-file-operators.htm)

|  |  |  |
| --- | --- | --- |
| 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. | [ -c $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. |

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](https://www.tutorialspoint.com/unix/if-fi-statement.htm)
* [if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)
* [if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm)

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](https://www.tutorialspoint.com/unix/case-esac-statement.htm)

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.

**IF –FI statement**

$ cat scrt1

#!/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

$ chmod +x scrt1

$ ls -lrt

total 4

-rwxr-xr-x 1 sm70471 gpa 127 Nov 23 22:44 scrt1

$ sh scrt1

a is not equal to b

$

**Function**

#!bin/sh

# Script for a function

add\_two\_num()

{

echo "ENTER FIRST NUMBER"

read a

echo "ENTER SECOND NUMBER"

read b

c=`expr $a + $b`

echo "THE SUM IS " $c

}

Execute the script as below

../file2.sh

bash-4.1$ . ./file2.sh

bash-4.1$ add\_two\_num

ENTER FIRST NUMBER

30

ENTER SECOND NUMBER

40

THE SUM IS 70

bash-4.1$ add\_two\_num

ENTER FIRST NUMBER

400

ENTER SECOND NUMBER

8204

THE SUM IS 8604

To get factorial of a number

10! = 10\*9\*8\*7\*6\*5\*4\*3\*2\*1=3628800

#!bin/sh

#Test to get the factorial of a number

Echo “ENTER A NUMBER”

Read n

I=1

Fact=1

For i<n

do

Fact=fact\*i

I++

Done

Echo “Factiorial of $n = “, $Fact