<https://www.tutorialspoint.com/unix/index.htm>

<https://www.tutorialspoint.com/unix_terminal_online.php>  
https://ryanstutorials.net/linuxtutorial/piping.php

<https://chmodcommand.com/chmod-755/> - CHMOD

**What is Unix ?**

The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the **operating system** or the **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 Architecture - Basic block diagram of Unix system –**



The main concept that unites all versions of Unix are following four basics −

* **Kernel** − The kernel is the heart of the operating system. It interacts with the 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 the most famous shells which are available with most of the Unix variants.
* **Commands and Utilities** − There are various commands and utilities which you can make use of 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 options.
* **Files and Directories** − All data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **filesystem**.

**Log in:**

* Give User Id (Case sensitive) and password (Case sensitive).
* Once logged in, we will be provided with a command prompt (sometime called the **$** prompt) where we type all commands.

login : amrood

amrood's password:

Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73

$

* To check calendar, you need to type the **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:**

* **Step 1** − Type **password** or **passwd** at the command prompt as shown below.
* **Step 2** − Enter old password, the one you're currently using and then type new password.
* **Step 3** − 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

$

**Who Are You?**

While you're logged into the system, if you want to know: Who am I? Enter the **whoami** command −

$ whoami

amrood

$

**Who is Logged in?**

To know who is logged in to the computer at the same time. There are three commands available to get you this information, based on how much you wish to know about the other users: **users**, **who**, and **w (**lists down information associated with the users logged in the system**)**.

$ users

amrood bablu qadir

$ who

amrood ttyp0 Oct 8 14:10 (limbo)

bablu ttyp2 Oct 4 09:08 (calliope)

$

**Logging Out**

When you finish your session, you need to log out of the system. To log out, type the **logout** command at the 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 system using predefined scripts to sync & clean system prior to shutting down |
| init 6 | Reboots the system by shutting it down completely and then restarting it |
| poweroff | Shuts down the system by powering off |
| reboot | Reboots the system |
| shutdown | Shuts down the system |

In Unix, there are three basic types of files −

* **Ordinary Files** − It is a file on the system that contains data, text, or program instructions.
* **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, 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 files & directories stored in the current directory, use command − **$ls**. The command **ls** supports the **-l** option which helps to get more information about listed files.

$ls

bin hosts lib res.03

ch07 hw1 pub test\_results

ch07.bak hw2 res.01 users

docs hw3 res.02 work

$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

-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

drwxr-xr-x 11 amrood amrood 4096 May 29 2007 zlib-1.2.3

$

* **First Column** − Represents the file type and the 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 the owner of the file. This is the Unix user who created this file.
* **Fourth Column** − Represents the group of owner. Every Unix user have an associated group.
* **Fifth Column** − Represents the file size in bytes.
* **Sixth Column** − Represents date & time when this file was created/ modified for the last time.
* **Seventh Column** − Represents the file or the directory name.

In the **ls -l** listing example, every file line begins with a **d**,**-**, or**l**. These characters indicate the type of the 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 a special meaning in Unix. For example, **\*** and **?** are meta characters. We use **\*** to match 0 or more characters, a question mark (**?**) matches with a single character.

$ls ch\*.doc

Displays all the files, the names of which start with ch and end 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 all the files ending with just .doc, can use the following command − **$ls \*.doc**

**Hidden Files**

For invisible files, the first character is the dot/ period (.). Unix programs (including the shell) use most of these files to store configuration information. Some examples of hidden 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 the invisible files, specify **ls −a**

$ ls -a

. .profile docs lib test\_results

.. .rhosts hosts pub users

.emacs bin hw1 res.01 work

.kshrc ch07.bak hw3 res.03

$

**Single dot (.)** represents the current directory and **Double dot (..)** represents the parent directory.

**Creating Files**

The **vi** editor is used to create ordinary files on any Unix system. Syntax: **$ vi filename**It will open a file with the given filename. Now, press the key “**I**” to come into edit mode. After that, you can start writing your content in the file as in the following program −

This is unix file....I created it for the first time.....

Once writing is done, press the key **esc** to come out of edit mode and then press **Shift + ZZ** together to come out of the file completely. File will be created with **filename** in the current directory.

**Editing Files**

You can edit an existing file using the **vi** editor. Syntax: **$ vi filename**Once the file is opened, come in edit mode by pressing key **i,** then by edit the file. For navigating inside a file, first come out of edit mode by using key **Esc**. Then use following keys −

|  |  |
| --- | --- |
| Keys | Function |
| l | to move to the right side |
| h | to move to the left side |
| k | to move upside in the file |
| j | to move downside in the file |

Use above keys to position the cursor where edit is required. Once positioned, use **i**to come in edit mode then do the editing then press **Esc** and finally **Shift + ZZ** together to come out of the file.

* Command “more” and “less”
* Run any command in background, use ‘&’ at last of the statement. Example: **$ls –l ch\*.doc &**
* In vim, “**dd**” is used to delete the line the cursor is on. You must be in command mode to use the commands. (Press Esc twice at any time to ensure that you are in command mode.)
* The vi editor has two kinds of searches: **string** & **character**. For a string search, the **“/”** (searches forwards / downwards in the file) and **“?”** (searches backwards / upwards in the file ).   
  When you start these commands, the command just typed will be shown on the last line of the screen, where you type the particular string to look for
* The substitution command (**:s/**) enables you to quickly replace words or groups of words within your files. Following is the syntax to replace text −  
  The **“g”** stands for globally. The result of this command is that all occurrences on the cursor's line are changed.

:s/search/replace/g

**Display Content of a File**

**cat** command is used to see the content of a file. You can display the line numbers by using the **-b** option along with the **cat** command as −

$ 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**

**wc** command is used to get a count of the total number of lines, words, and characters in a file (or multiple files) .File created and below is the detail of all the 4 columns:

$ wc filename

2 19 103 filename

$

* **First Column** − Represents the total number of lines in the file.
* **Second Column** − Represents the total number of words in the file.
* **Third Column** − Represents the total number of bytes in a file. This is the actual size of file.
* **Fourth Column** − Represents the file name.

$ wc filename1 filename2 filename3

**Copying Files**

To make a copy of a file use the **cp** command. The basic syntax − **$ cp source\_file destination\_file**

**Renaming Files**

To change the name of a file, use the **mv** command. Following is the basic syntax − **$ mv old\_file new\_file**The **mv** command move the existing file into the new file. Here, you will find only **newfile** in your current directory.

**Deleting Files**

To delete an existing file (can delete multiple files too), use the **rm** command. Following is the basic syntax − **$ rm filename1 filename2  
Caution** − It is always recommended to be careful while using this **Delete** command. It is better to use the **-i** option along with **rm** command (will ask for confirmation).

**Standard Unix Streams**

Normally, every Unix program has three streams (files) opened for it when it starts up −

* **stdin**  − This is referred to as the *standard input* and the associated file descriptor is 0. This is also represented as STDIN. The Unix program will read the default input from STDIN.
* **stdout** − This is referred to as the *standard output* and the associated file descriptor is 1. This is also represented as STDOUT. The Unix program will write the default output at STDOUT
* **stderr** − This is referred to as the *standard error* and the associated file descriptor is 2. This is also represented as STDERR. The Unix program will write all the error messages at STDERR.

**Home Directory**

Unix uses a hierarchical structure for organizing files & directories. This structure is referred as a directory tree with a single root node, a slash (/) and all other directories are contained below it. The directory initiated during first login is home directory. You will do much work in home directory and subdirectories that you'll be creating to organize your files. You can use the following commands:

|  |  |
| --- | --- |
| Commands | Function |
| **$cd ~** | to go in your home directory |
| **$cd ~username** | to go in any other user's home directory |
| **$cd -** | to go in your last directory |

**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 pathname is absolute, if it is described in relation to root. Absolute pathnames always begin with a **/**. Examples of absolute filenames.

/etc/passwd

/users/sjones/chem/notes

A pathname can also be relative to your current working directory. Relative pathnames never begin with **/**. Relative to user amrood's 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, **$ls dirname** is used. 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.   
Another example − **$mkdir /tmp/test-dir** that creates the directory **test-dir** in the **/tmp** directory. The **mkdir** command produces no output if it successfully creates the requested directory.  
If you give more than one directory on the command line, **mkdir** creates each of the directories. Example, it creates the directories docs and pub under the current directory − **$mkdir docs pub**

**Creating Parent Directories**

Now understand how to create parent directories. Sometimes if we want to create a directory & its parent directory / directories may 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 **mkdir -p** command. It creates all the necessary directories for you. For example, below command creates all the required parent directories −

$mkdir -p /tmp/amrood/test

$

**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. Below command removes directories dirname1, dirname2 & dirname3, if they are empty. The **rmdir** command produces no output if it is successful.

$rmdir dirname1 dirname2 dirname3

$

**Changing Directories**

Use **cd** command to do more than just change to a home directory. You can use it to change to any directory by giving a valid absolute or relative path. The syntax is given below − **$cd dirname**Here, **dirname** is the name of the directory that you want to change to. For example, the command − **$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)**

**filename .** (dot) represents the current working directory; and **filename ..** (dot-dot) represents the directory one level above the current working directory, often referred to as the parent directory.  
Entering the command to show a listing of the current working directories/files & use the **-a option** to list all the files and the **-l option** to provide the long listing, we will receive the following 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 permission/ Access mode**

Every file in Unix has the following attributes −

* **Owner permissions** − It determines what actions the owner of the file can perform on the file.
* **Group permissions** − It determines what actions a user, who is a member of the group that a file belongs to, can perform on the file.
* **Other (world) permissions** − It indicates what action all other users can perform on the file.

**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 permissions are the read, write and execute permissions.

**Read:** Grants the capability to read, i.e., view the contents of the file.

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

### **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 −

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

**Write:** Access means that the user can add or delete files from the directory.

**Execute:** Executing a directory doesn't really make sense, so think of this as a traverse permission (x means searchable when it comes to directory).

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

**Changing Permissions**

**Using chmod in Symbolic Mode:**The easiest way to modify file / directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete or specify the permission set.

|  |  |
| --- | --- |
| **S.No.** | **Chmod operator & Description** |
| 1 | **+ (**Adds the designated permission(s) to a file or directory**)** |
| 2 | -      **(**Removes the designated permission(s) from a file or directory.**)** |
| 3 | **= (**Sets the designated permission(s).**)** |

|  |  |  |
| --- | --- | --- |
| Results after ls -l testfile | Commands | Group |
| rwx rwx r | ls -l testfile |  |
| rwx rwx rwx | chmod o+wx testfile | others - o |
| rw rwx rwx | chmod u-x testfile | user - u |
| rw rx rwx | chmod g = rx testfile | group - g |

**Using chmod with Absolute Permissions:**

|  |  |  |
| --- | --- | --- |
| 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 |

|  |  |
| --- | --- |
| Results after ls -l testfile | Commands |
| rwx-rwx-r | ls -l testfile |
| rwxr-xr-x | chmod 755 testfile |
| rwxr---wx | chmod 743 testfile |
| ----r---wx | chmod 043 testfile |

**Changing Owners and Groups**

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

Value of the user can be either the **name of a user** or the **user id (uid)** of a user on the system. Example, it changes the owner of the given file to the user **amrood** − **$ chown amrood testfile**  
**Changing Group Ownership -** The **chgrp** command changes the group ownership of a file. The basic syntax is as follows − **$ chgrp group filelist**

Value of the group can be either the **name of a group** or  **group ID (GID)** of a group on the system. Example, it changes the group of the given file to **special** group − **$ chgrp special testfile**

**NOTE −** the super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

**SUID and SGID File Permission**

Often when a command is executed, it has to be executed with special privileges to accomplish its task. Example, when password is changed with **passwd** command, new password is stored in the file **/etc/shadow**. A regular user do not have **read** or **write** access to this file for security reasons, but when he changes password, he should have the write permission to this file. This means that the **passwd** program has to give him additional permissions so that he 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. Executing a program that has SUID bit enabled, inherits the permissions of that program's owner. Programs which don’t have SUID bit set are run with permission of the user who started the program.

This is the case with 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 & 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 normally resides.

$ ls -l /usr/bin/passwd

-r-sr-xr-x 1 root bin 19031 Feb 7 13:47 /usr/bin/passwd\*

$

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 command −

$ chmod ug+s dirname

$ ls -l

drwsr-sr-x 2 root root 4096 Jun 19 06:45 dirname

$

'S' = The directory's setgid bit is set, but the execute bit isn't set.  
's' = The directory's setgid bit is set, and the execute bit is set.  
SetGID = If other user creates a file/directory under such a setgid directory, the new file/directory group is set as the group of the directory's owner, instead of the group of the user who created it.  
To remove the SetGID bit: chmod g-s file\_name / dir\_name

--------------------------------

"s", like "x", means something different for directories and regular files. For files, "x" means "executable" of course. For directories, it means "searchable." Without "x" permission on a directory, you can't set it to be your current directory, or get any of the file information like size, permissions, so that you effectively can't access any of the files. If a directory has no "r" permission, you can't get a listing, but if you know a file is there, you can still access the file.  
  
Now "s", for files, means "setuid exec." If a file has s permission, then it's executable, and furthermore, the user id and/or group id of the process is set to the user or group id of the owner of the file, depending on whether it's the user or group "s" that's set. This is a way to give limited root powers to a user -- a program that runs as root when an ordinary user executes it. For example, the "passwd" program, which can change otherwise write-protected files on behalf of a user, works this way: it's owned by the "bin" group (generally) and has g+s so that it can write to /etc/passwd and/or /etc/opasswd which are also owned by group "bin."  
For directories, "s" means "sticky". If a directory has "s", then the owner and/or group of any files put into the directory are set to the owner/group of the directory. This is often used on CVS repositories, so that the files in the repository end up all owned by the same person and/or group, even though they're put in by different people. I use g+s on all the CVS repositories I set up.

Chmod 4655 file1 : -rwSr-xr-x  
Chmod 4755 file1 : -rwsr-xr-x

**ECHO**

$TEST="Unix Programming"  
$echo $TEST

It produces the following result: **Unix Programming**

**Initialization**  
After logging the system, the shell undergoes a phase **Initialization** to set up the environment. It is usually a 2-step process involving shell to read below files −

* **/etc/profile**: The shell checks if the file **/etc/profile** exists. If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.  
  The file **/etc/profile** is maintained by the system admin of your Unix machine and contains shell initialization information required by all users on a system.
* **profile**: The shell checks if the file **.profile** exists in your home directory, the one that starts out in after you log in. If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.  
  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 the commands.  
  - A list of variables affecting the look and feel of your terminal. **.profile** will be available in your home directory. Open it using the **vi** editor and check all the variables set for your environment.

As soon as both of these files have been read, the shell displays a prompt − $  
**Note**: The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

**Setting the Terminal Type**

Usually, the type of terminal used is automatically configured by the **login** or **getty** programs. Sometimes, auto configuration process guesses the terminal incorrectly and terminal is set incorrectly, leading the output of the commands to look strange, or you will be unable to interact with the shell properly. To assure this does not happen, set the terminal to the lowest common denominator.

$TERM=vt100

$

**Setting the PATH**  
Typing any command makes the shell to locate the command before its execution. The **PATH** variable specifies the locations in which the shell should look for commands. Usually the Path variable is set as follows − **$PATH=/bin:/usr/bin**Here, each of the individual entries separated by the colon (:) are directories. If a command is executed and it is not found in any of directories given in the PATH variable, a message appears −

$hello  
hello: not found  
$

**PS1 and PS2 Variables**The characters that the shell displays as your command prompt are stored in the variable PS1. This variable to be anything. As soon as you change it, it'll be used by the shell from that point on.

$PS1='=>'  
=>

Your prompt will 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.

|  |  |
| --- | --- |
| S.No. | 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 when a new command is 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 $ sign |

You can 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 an incomplete command, the shell will display a secondary prompt and wait to complete the command and **Enter** again. The default secondary prompt is **>** (greater than), but can be changed by re-defining the **PS2** shell variable. Example:

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

The example given below re-defines 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 are set and accessed as mentioned below −

|  |  |
| --- | --- |
| S.No. | Escape Sequence & 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 | A Unix system with a dynamic linker, contains a colon separated list of directories that dynamic linker should search for shared objects when building a process image after exec, before searching in other directories. |
| PATH | Indicates the 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 the 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  
$

**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 −s

|  |  |
| --- | --- |
| S.No. | Option & Description |
| -s | Specifies subject on the command line. |
| -c | Sends carbon copies to the list of users. List should be a comma separated list of names. |
| -b | Sends blind carbon copies to list. List should be a comma separated list of names. |

Following is an 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 "**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

**Redirecting:**

Every program we run on the command line automatically has three data streams connected to it.  
\* STDIN (0) - Standard input (data fed into the program)  
\* STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)  
\* STDERR (2) - Standard error (for error messages, also defaults to the terminal)

**To a file:**

Normally, we will get our output on the screen, which is convenient, but sometimes we may wish to save it into a file to keep as a record, feed into another system, or send to someone else. The greater than operator ( > ) indicates to the command line that we wish the programs output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen. Let's see an example.

$ **ls**$ barry.txt example.png firstfile foo1 video.mpeg  
$ **ls > myoutput**$ **ls**$ barry.txt example.png firstfile foo1 myoutput video.mpeg  
$ **cat myoutput**barry.txt  
example.png  
firstfile  
foo1  
myoutput  
video.mpeg  
$

**Note :** “$” is not appearing in cat myoutput, so it is useful for printing purpose. Newly created file will also come in report.

**Saving in an existing file:**We can instead get the new data to be appended to the file by using the double greater than operator ( >> ).

$ **cat myoutput**$ barry.tx  
$ l**s >> myoutput**$ **cat myoutput**barry.txt  
barry.txt  
example.png  
firstfile  
myoutput  
video.mpeg  
$

**Redirecting from a file:**If we use the less than operator ( < ) then we can send data the other way. We will read data from the file and feed it into the program via it's STDIN stream.  
**Note:** File name is not appearing in cat myoutput, so it is useful for printing purpose.

$ **wc -l myoutput**8 myoutput  
$ **wc -l < myoutput**8  
$

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

$ **wc -l < barry.txt > myoutput**$ **cat myoutput**7  
$

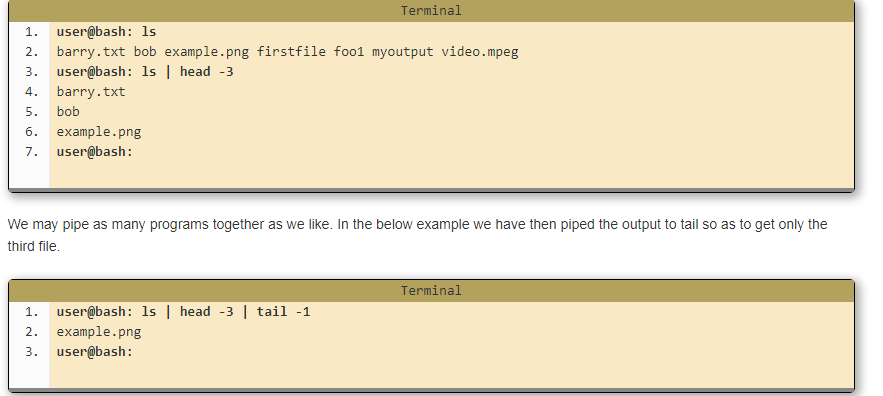
**Redirecting STDERR (Space matters):**Now the 3rd stream which is Standard Error or STDERR. The 3 streams actually have numbers associated with them. STDERR is stream number 2 and we may use these numbers to identify the streams. If we place a number before the > operator then it will redirect that stream (if we don't use a number, like we have been doing so far, then it defaults to stream 1).

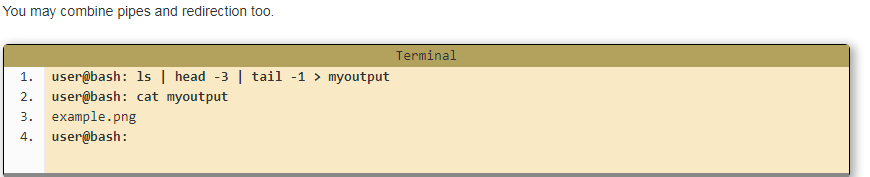
$ **ls -l video.mpg blah.foo**ls: cannot access blah.foo: No such file or directory  
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg  
$ **ls -l video.mpg blah.foo 2> errors.txt**-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg  
$ **cat errors.txt**ls: cannot access blah.foo: No such file or directory  
$

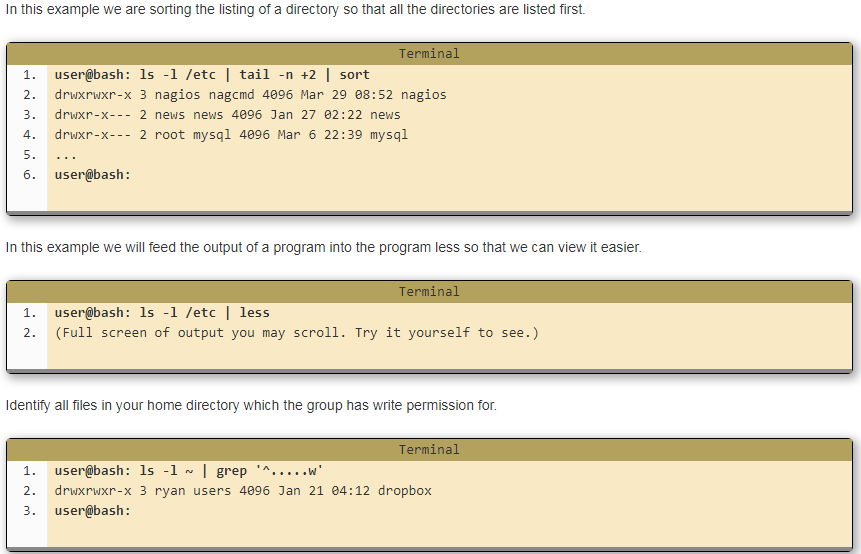
Maybe we wish to save both normal output and error messages into a single file. This can be done by redirecting the STDERR stream to the STDOUT stream and redirecting STDOUT to a file. We redirect to a file first then redirect the error stream. We identify the redirection to a stream by placing ‘&‘ in front of the stream number (otherwise it would redirect to a file called 1).

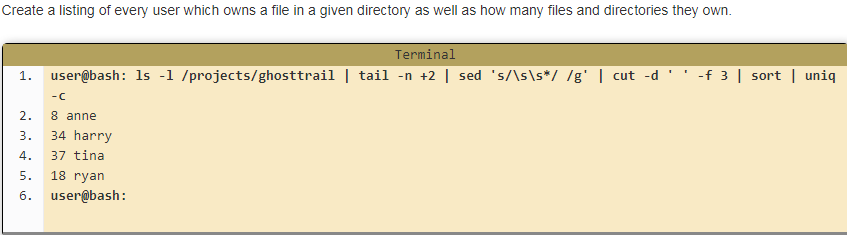
$ **ls -l video.mpg blah.foo > myoutput 2>&1**$ **cat myoutput**ls: cannot access blah.foo: No such file or directory  
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg  
$

**Piping: (Space matters)**

Piping is a mechanism for sending data from one program to another. It uses the operator ( | ) (above ( \ ) key on keyboard). This operator feeds the output from the program on the left as input to the program on the right. In the example below we will list only the first 3 files in the directory.  
  








**The grep Command**  
$ls -l | grep "Aug"  
Lists all files with “Aug” mentioned in the line.

|  |  |
| --- | --- |
| **grep Option** | **Descriptions** |
| -v | Prints all lines that do not match pattern. |
| -n | Prints the matched line and its line number. |
| -l | Prints only the names of files with matching lines (letter "l") |
| -c | Prints only the count of matching lines. |
| -i | Matches either upper or lowercase. |

**Sort Options**

This command does not actually change the input file.

<https://www.geeksforgeeks.org/sort-command-linuxunix-examples/>

|  |  |
| --- | --- |
| **sort Option** | **Descriptions** |
| -n | Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs. |
| -r | Reverses the order of sort. |
| -f | Sorts upper and lowercase together. |
| +x | Ignores first x fields when sorting. |

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

$

$ls -l | grep "Aug" | sort –r –k 5 #### correct and practiced

-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

$

$ls -l | grep "Aug" | sort –r –k 5 | tail -2 #### correct and practiced #### ls -l | grep "Aug" | sort –r –k5 | tail -2

-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02

$

$cat file2

A 1

A 2

B 3

B 2  
B 4

C 1

C 5

D 4

D 3

E 2

$sort -k1,1 -u file2

A 1

B 3

C 1

D 4

E 2

**Fetching unique records:**

File f1 has below data :  
**> cat file1**

abc 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452  
abc 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452  
tas 3420 3562 2123 1343 2176 7654 3252 8765 5643 3452  
aer 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452  
tas 3420 3562 2123 1343 2176 7654 3252 8765 5643 3452

Below will display the number of occurrence and the record

**> sort file1 | uniq -c**

2 abc 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452

1 aer 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452

2 tas 3420 3562 2123 1343 2176 7654 3252 8765 5643 3452

Below will display only the duplicate records

**> sort file1 | uniq -d**

abc 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452

tas 3420 3562 2123 1343 2176 7654 3252 8765 5643 3452

Below will display distinct records

**> sort file1 | uniq**

abc 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452

aer 1000 3452 2463 2343 2176 7654 3452 8765 5643 3452

tas 3420 3562 2123 1343 2176 7654 3252 8765 5643 3452

**Listing Running Processes**

* To see the process status, command **: $ps (or $ps –f for full details)**

$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

|  |  |
| --- | --- |
| **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 |

|  |  |
| --- | --- |
| **Tags** | **Description** |
| -a | Shows information about all users |
| -x | Shows information about processes without terminals |
| -u | Shows additional information like -f option |
| -e | Displays extended information |

* To kill any process, **CTRL + C or kill -9 process id**

**The ftp Utility**Here, ftp stands for **File Transfer Protocol**. This utility helps you upload and download your file from one computer to another computer.

<https://linuxroutes.com/3-ways-check-open-ports-linux-respective-service-process/> - for checking opened ports

$ftp hostname or ip-address

$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

-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

-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

ftp> put wave\_shift1

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**

There are times when we are required 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 login using Telnet, you can perform all the activities on your remotely connected machine. Example of 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:>

**Simple shell script (Bourne shell)**

Bourne shell is usually installed as /bin/sh on most versions of Unix. We create a **test.sh** script. **Note,** all the scripts would have the **.sh** extension. Firstly, we need to alert the system that a shell script is being started. It is done using the **shebang** construct. It tells the system that commands that follow are to be executed by the Bourne shell. *It's called a shebang as the****#****symbol is called a hash, and the* ***!*** *is called a bang*. To create a script containing these commands, you put the shebang line first and then add the commands −

#!/bin/bash  
# Author : Zara Ali  
pwd  
ls

Save the above content and make the script executable −

$chmod +x test.sh

The shell script is now ready to be executed −

$./test.sh

Upon execution, you will receive the following result −

/home/amrood  
index.htm unix-basic\_utilities.htm unix-directories.htm   
test.sh unix-communication.htm unix-environment.htm

**Note** − To execute a program available in the current directory, use **./program\_name**

#!/bin/sh  
# Author : Zara Ali  
echo "What is your name?"  
read PERSON  
readonly NAME  
NAME="Qadiri"  
echo "Hello, $PERSON and $NAME"  
unset NAME  
echo “Hellow, $NAME” --🡪 does not print anything as the variable has been unset

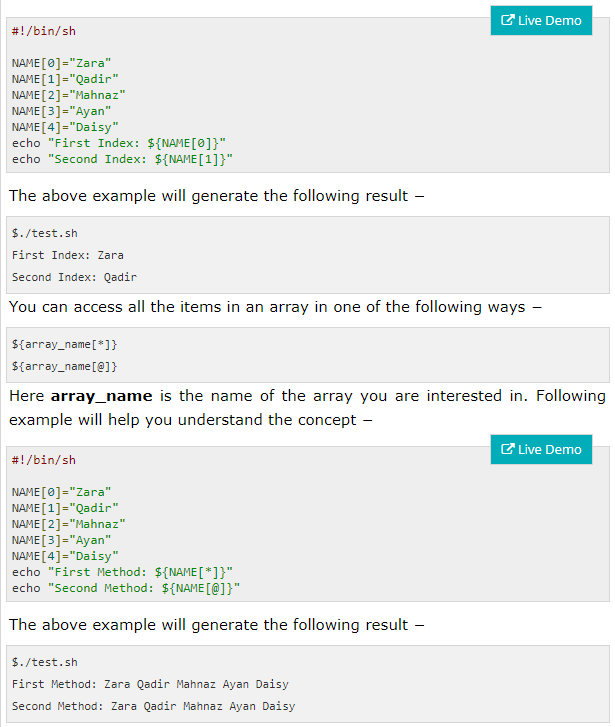
Here is a sample run of the script −

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

Special Variables: <https://www.tutorialspoint.com/unix/unix-special-variables.htm>

**Arrays:**

array\_name[index]=value



Bourne shell don't have any mechanism to perform simple arithmetic operations. It uses external programs, either **awk** or **expr**. Example to add two numbers −

[Live Demo](http://tpcg.io/zURE2C)

#!/bin/sh  
val=`expr 2 + 2`  
echo "Total value : $val"

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

* There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* The complete expression should be enclosed between **‘ ‘**, called the backtick.

**Arithmetic Operators**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| #NAME? | Assigns 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 important to know that **all conditional expressions** should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.