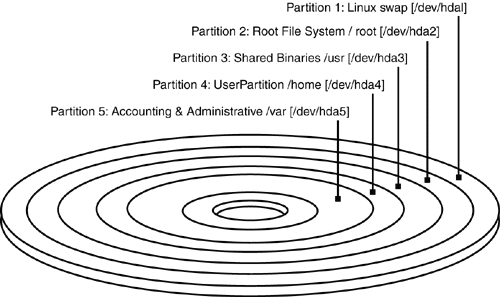
**Overview of file system**

A **filesystem** is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk, that helps in efficient storage and retrieval of data. The basic file system for UNIX is **UNIX File System, or UFS**, or sometimes called the Berkeley Fast File system. It introduced the concept of inodes which contain the metadata that describes a file



EACH PARTITION WILL CONTAIN FILES. EACH FILE HAS INODE INFORMATION WITH ATTRIBUTES AND BLOCK ADDRESSES TO LOCATE FILES.

EACH PARTITION WILL HAVE ITS OWN INODE BLOCK.

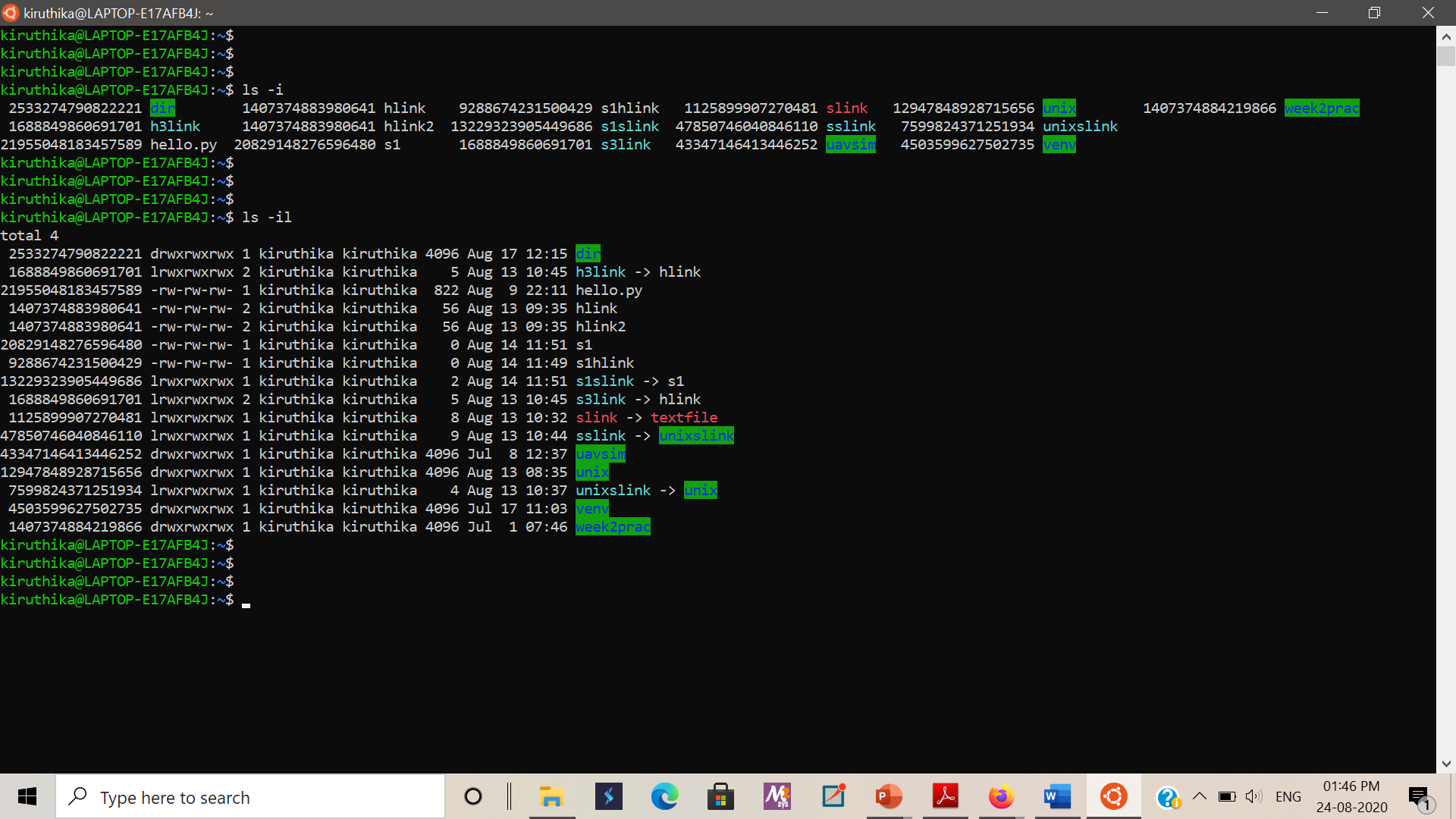
ALL THE INODE INFORMATION OF FILES PRESENT IN A PARTITION WILL BE AVAILABLE IN INODE BLOCK OF THAT PARTITION.

A Unix file is a store house of information. It is simply a sequence of bytes. Files are divided into these categories.

**File Types**

The UNIX filesystem contains several different types of files:

* Ordinary Files
  + Used to contain only data, source programs, Unix commands as well as any files created by user.
  + This is the type of file that you usually work with.
* Directories
  + Branching points in the hierarchical tree
  + Used to organize groups of files
  + May contain ordinary files, special files or other directories
  + Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
  + All files are descendants of the root directory, ( named / ) located at the top of the tree.
  + They usually map the filenames they contain with their respective inode number



* Symbolic link
  + A symbolic link is a reference to another file. This special file is stored as a textual representation of the referenced file's path (which means the destination may be a relative path, or may not exist at all).
  + A symbolic link is marked with an l (lower case L) as the first letter of the mode string, e.g.

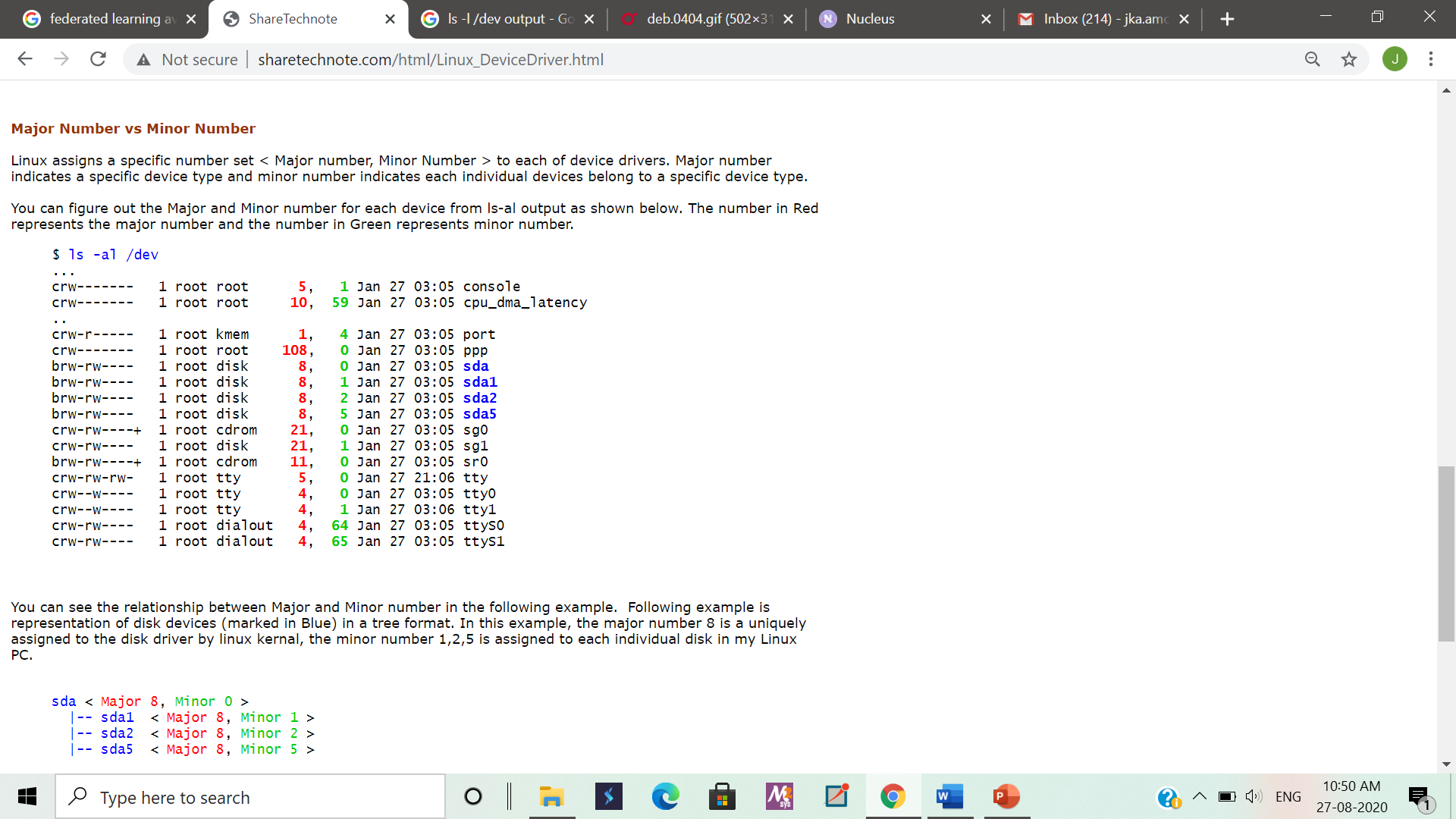
lrwxrwxrwx ... termcap -> /usr/share/misc/termcap

lrwxrwxrwx ... S03xinetd -> ../init.d/xinetd

* + Symbolic links are depicted in the above screenshot.
* Special Files / Device files
  + Device files allow user programs to access hardware devices on the system through the kernel.
  + you can read from them, write to them
  + When you access such a device "file," the kernel recognizes the I/O request and passes it to a **device driver** with some parameters, which performs some operation, such as reading data from a serial port, or sending data to a sound card.
    - The routines for a specific device are known as device drivers
  + Device files provide a convenient way to access system resources without requiring the applications programmer to know how the underlying device works. Under Linux, as with most Unix systems, device drivers themselves are part of the kernel.
  + Device files are located in the directory /dev on nearly all Unix-like systems. Each device on the system should have a corresponding entry in /dev.
    - For example, /dev/ttyS0 corresponds to the first serial port,; /dev/hda2 corresponds to the second partition on the first IDE drive.
    - In fact, there should be entries in /dev for devices you do not have. The device files are generally created during system installation and include every possible device driver. They don't necessarily correspond to the actual hardware on your system.
  + When using ls -l to list device files in /dev, you'll see something like the following:

brw-rw---- 1 root disk **3, 0** May 19 1994 /dev/hda

* + This is /dev/hda, which corresponds to the first IDE drive. First of all, note that the first letter of the permissions field is b, which means this is a block device file. (Recall that normal files have a - in this first column, directories a d, and so on.) Device files are denoted either by b, for block devices, or c, for character devices.
    - A block device is usually a peripheral such as a hard drive: data is read and written to the device as entire blocks (where the block size is determined by the device; it may not be 1024 bytes as we usually call "blocks" under Linux), and the device may be accessed randomly.
    - In contrast, character devices are usually read or written sequentially, and I/O may be done as single bytes. An example of a character device is a serial port.
  + Also, note that the size field in the ls -l listing is replaced by two numbers, separated by a comma. The first value is the **major device number** and the second is the **minor device number**. When a device file is accessed by a program, the kernel receives the I/O request in terms of the major and minor numbers of the device.
    - The major number generally specifies a particular driver within the kernel, and the minor number specifies a particular device handled by that driver.
    - For example, all serial port devices have the same major number, but different minor numbers. The kernel uses the major number to redirect an I/O request to the appropriate driver, and the driver uses the minor number to figure out which specific device to access



* + The great exception is network devices, which do not turn up in the file system (no device files) but are handled separately.

POSIX standard includes Sockets and Named Pipes as file types too.

**File Names**

UNIX permits file names to use most characters, but avoid spaces, tabs and characters that have a special meaning to the shell, such as:

**& ; ( ) | ? \ ' " ` [ ] { } < > $ - ! /**

Case Sensitivity: uppercase and lowercase are not the same! These are three different files:

**NOVEMBER November november**

Length: can be up to 256 characters

Extensions: may be used to identify types of files

**program.c  *- C language source file***

**alpha2.f  *- Fortran source file***

**xwd2ps.o  *- Object/executable code***

**mygames.Z  *- Compressed file***

Hidden Files: have names that begin with a dot (.) For example: ls -ali

**.cshrc .login .mailrc .mwmrc**

Uniqueness: as children in a family, no two files with the same parent directory can have the same name. Files located in separate directories can have identical names.

Reserved Filenames:

**/ *- the root directory (slash)***

**. *- current directory (period)***

**.. *- parent directory (double period)***

**~ *- your home directory (tilde)***

# Manage file permissions on Unix-like systems

## **Overview**

[Unix](https://kb.iu.edu/d/agat)-like operating systems, such as Linux, running on shared high-performance computers use settings called permissions to determine who can access and modify the files and directories stored in their file systems. Each file and directory in a file system is assigned "owner" and "group" attributes.

Most commonly, by default, the user who creates a file or directory is set as owner of that file or directory. When needed (for example, when a member of your research team leaves), the system's root administrator can change the user attribute for files and directories.

The group designation can be used to grant teammates and/or collaborators shared access to an owner's files and directories, and provides a convenient way to grant access to multiple users.

## **View file permissions**

To view the permissions for all files in a directory, use the ls command with the -l option.

For example, if you enter:

ls -l

You should see output similar to the following:

-rw-r--r-- 1 user1 group1 62 Jan 15 16:10 myfile.txt

drwxr-xr-x 2 user1 group1 2048 Jan 15 17:10 Example

In the output example above, the first character in each line indicates whether the listed object is a file or a directory. Directories are indicated by a (d); the absence of a d at the beginning of the first line indicates that myfile.txt is a regular file.

The letters rwx represent different permission levels:

| **Permission** | **Files** | **Directories** |
| --- | --- | --- |
| r | can read the file | can ls the directory |
| w | can write the file | can modify the directory's contents |
| x | can execute the file | can cd to the directory |

Note the multiple instances of r, w, and x. These are grouped into three sets that represent different levels of ownership:

* **Owner or user permissions:** After the directory (d) slot, the first set of three characters indicate permission settings for the owner (also known as the user).

In the example -rw-r--r--, the owner permissions are rw-, indicating that the owner can read and write to the file but can't execute it as a program.

In the example drwxr-xr-x, the owner permissions are rwx, indicating that the owner can view, modify, and enter the directory.

* **Group permissions:** The second rwx set indicates the group permissions. In the fourth column of the example above, group1 is the group name.

In the example -rw-r--r--, group members can only read the file.

In the example drwxr-xr-x, group members can view as well as enter the directory.

* **Other permissions:** The final rwx set is for "other" (sometimes referred to as "world"). This is anyone outside the group. In both examples above, these are set to the same permissions as the group.

## **Change file permissions**

To change file and directory permissions, use the command chmod (change mode). The owner of a file can change the permissions for user (u), group (g), or others (o) by adding (+) or subtracting (-) the read, write, and execute permissions.

There are two basic ways of using chmod to change file permissions: The symbolic method and the absolute form.

### **Symbolic method**

The first and probably easiest way is the relative (or symbolic) method, which lets you specify permissions with single letter abbreviations. A chmod command using this method consists of at least three parts from the following lists:

| **Access class** | **Operator** | **Access Type** |
| --- | --- | --- |
| u (user) | + (add access) | r (read) |
| g (group) | - (remove access) | w (write) |
| o (other) | = (set exact access) | x (execute) |
| a (all: u, g, and o) |  |  |

For example, to add permission for everyone to read a file in the current directory named myfile, at the Unix prompt, enter:

chmod a+r myfile

The a stands for "all", the + for "add", and the r for "read".

**Note:**

This assumes that everyone already has access to the directory where myfile is located and its parent directories; that is, you must set the directory permissions separately.

If you omit the access class, it's assumed to be all, so you could also enter the previous example as:

chmod +r myfile

You can also specify multiple classes and types with a single command. For example, to remove read and write permission for group and other users (leaving only yourself with read and write permission) on a file named myfile, you would enter:

chmod go-rw myfile

You can also specify that different permissions be added and removed in the same command. For example, to remove write permission and add execute for all users on myfile, you would enter:

chmod a-w+x myfile

In each of these examples, the access types that aren't specified are unchanged. The previous command, for example, doesn't change any existing settings specifying whether users besides yourself may have read (r) access to myfile. You could also use the exact form to explicitly state that group and other users' access is set only to read with the = operator:

chmod go=r myfile

The chmod command also operates on directories. For example, to remove write permission for other users on a subdirectory named mydir, you would enter:

chmod o-w mydir

To do the same for the current directory, you would enter:

chmod o-w

To change permissions recursively in all subdirectories below the specified directory, add the -R option; for example, to grant execution permissions for other users to a directory (mydir) and all the subdirectories it contains, you would enter:

chmod -R o+x mydir

Be careful when setting the permissions of directories, particularly your home directory; you don't want to lock yourself out by removing your own access. Also, you must have execute permission on a directory to switch (cd) to it.

### **Absolute form**

The other way to use the chmod command is the absolute form, in which you specify a set of three numbers that together determine all the access classes and types. Rather than being able to change only particular attributes, you must specify the entire state of the file's permissions.

The three numbers are specified in the order: user (or owner), group, and other. Each number is the sum of values that specify read, write, and execute access:

| **Permission** | **Number** |
| --- | --- |
| Read (r) | 4 |
| Write (w) | 2 |
| Execute (x) | 1 |

Add the numbers of the permissions you want to give; for example:

* For file myfile, to grant read, write, and execute permissions to yourself (4+2+1=7), read and execute permissions to users in your group (4+0+1=5), and only execute permission to others (0+0+1=1), you would use:

chmod 751 myfile

* To grant read, write, and execute permissions on the current directory to yourself only, you would use:

chmod 700

You can think of the three digit sequence as the sum of attributes you select from the following table:

|  |  |
| --- | --- |
| Read by owner | 400 |
| Write by owner | 200 |
| Execute by owner | 100 |
| Read by group | 040 |
| Write by group | 020 |
| Execute by group | 010 |
| Read by others | 004 |
| Write by others | 002 |
| Execute by others | 001 |

Sum all the accesses you wish to permit. For example, to give write and execute privileges to the owner of myfile (200+100=300), and give read privileges to all (400+040+004=444), you would enter:

chmod 744 myfile

**Some other examples are:**

|  |  |
| --- | --- |
| 777 | anyone can do anything (read, write, or execute) |
| 755 | you can do anything; others can only read and execute |
| 711 | you can do anything; others can only execute |
| 644 | you can read and write; others can only read |

**Permissions for directories:**

* The read (r) permission lets users **look** (ls) into directories.
* The execute (x) permission lets users **move** (cd) into directories.
* The write (w) permission lets users **add and remove** files.

# What are the Different Types of Shells in Linux?

Shells are an important part of any Linux user session. We are provided several different types of shells in Linux to accomplish tasks. Each shell has unique properties. Hence, there are many instances where one shell is better than the other for specific requirements.

This makes it important for us to be aware about the different types of shells available in Linux. In this tutorial, we will discuss what is a shell and why is it important.

Further, we will explore different types of shells in Linux to understand their functions and properties.

## What is a Shell, and Why do we need them?

Whenever a user logs in to the system or opens a console window, the kernel runs a new shell instance. The kernel is the heart of any operating system.

It is responsible for the control management, and execution of processes, and to ensure proper utilization of system resources.

A shell is a program that acts as an interface between a user and the kernel. It allows a user to give commands to the kernel and receive responses from it. Through a shell, we can execute programs and utilities on the kernel. Hence, at its core, a shell is a program used to execute other programs on our system.

Being able to interact with the kernel makes shells a powerful tool. Without the ability to interact with the kernel, a user cannot access the utilities offered by their machine’s operating system.

Let’s understand the major shells that are available for the Linux environment.

## Different Types of Shells in Linux

If you now understand what a kernel is, what a shell is, and why a shell is so important for Linux systems, let’s move on to learning about the different types of shells that are available.

Each of these shells has properties that make them highly efficient for a specific type of use over other shells. So let us discuss the different types of shells in Linux along with their properties and features.

### 1. The [Bourne Shell](https://en.wikipedia.org/wiki/Bourne_shell) (sh)

Developed at AT&T Bell Labs by Steve Bourne, the Bourne shell is regarded as the first UNIX shell ever. It is denoted as sh. It gained popularity due to its compact nature and high speeds of operation.

This is what made it the default shell for Solaris OS. It is also used as the default shell for all Solaris system administration scripts. Start reading about [shell scripting here](https://www.journaldev.com/35945/arrays-in-shell-scripts).

**However, the Bourne shell has some major drawbacks.**

* It doesn’t have in-built functionality to handle logical and arithmetic operations.
* Also, unlike most different types of shells in Linux, the Bourne shell cannot recall previously used commands.
* It also lacks comprehensive features to offer a proper interactive use.

The complete path-name for the Bourne shell is /bin/sh and /sbin/sh. By default, it uses the prompt # for the root user and $ for the non-root users.

### 2. The [GNU Bourne-Again Shell](https://www.gnu.org/software/bash/) (bash)

More popularly known as the Bash shell, the GNU Bourne-Again shell was designed to be compatible with the Bourne shell. It incorporates useful features from different types of shells in Linux such as Korn shell and C shell.

It allows us to automatically recall previously used commands and edit them with help of arrow keys, unlike the Bourne shell.

The complete path-name for the GNU Bourne-Again shell is /bin/bash. By default, it uses the prompt bash-VersionNumber# for the root user and bash-VersionNumber$ for the non-root users.

### 3. The [C Shell](http://bxr.su/NetBSD/bin/csh/) (csh)

The C shell was created at the University of California by Bill Joy. It is denoted as csh. It was developed to include useful programming features like in-built support for arithmetic operations and a syntax similar to the C programming language.

Further, it incorporated command history which was missing in different types of shells in Linux like the Bourne shell. Another prominent feature of a C shell is “aliases”.

The complete path-name for the C shell is /bin/csh. By default, it uses the prompt hostname# for the root user and hostname% for the non-root users.

### 4. The [Korn Shell](http://www.kornshell.com/) (ksh)

The Korn shell was developed at AT&T Bell Labs by David Korn, to improve the Bourne shell. It is denoted as ksh. The Korn shell is essentially a superset of the Bourne shell.

Besides supporting everything that would be supported by the Bourne shell, it provides users with new functionalities. It allows in-built support for arithmetic operations while offereing interactive features which are similar to the C shell.

The Korn shell runs scripts made for the Bourne shell, while offering string, array and function manipulation similar to the C programming language. It also supports scripts which were written for the C shell. Further, it is faster than most different types of shells in Linux, including the C shell.

The complete path-name for the Korn shell is /bin/ksh. By default, it uses the prompt # for the root user and $ for the non-root users.

### 5. The [Z Shell](https://ohmyz.sh/) (zsh)

The Z Shell or zsh is a sh shell extension with tons of improvements for customization. If you want a modern shell that has all the features a much more, the zsh shell is what you’re looking for.

**Some noteworthy features of the z shell include:**

* Generate filenames based on given conditions
* Plugins and theming support
* Index of built-in functions
* Command completion
* and many more…

Let us summarize the different shells in Linux which we discussed in this tutorial in the table below.

To see in which shell you are currently in use the command **echo $SHELL**

|  |  |  |  |
| --- | --- | --- | --- |
| **Shell** | **Complete path-name** | **Prompt for root user** | **Prompt for non root user** |
| Bourne shell (sh) | /bin/sh and /sbin/sh | # | $ |
| GNU Bourne-Again shell (bash) | /bin/bash | bash-VersionNumber# | bash-VersionNumber$ |
| C shell (csh) | /bin/csh | # | % |
| Korn shell (ksh) | /bin/ksh | # | $ |
| Z Shell (zsh) | /bin/zsh | <hostname># | <hostname>% |

## Wrapping up

Shells are one of, if not the most powerful tools available to a Linux user. Without shells, it is practically impossible for a person to utilise the features and functionality offered by the kernel installed on their system.

## **How to Change my default shell**

1. First, find out the available shells on your Linux box, run**cat /etc/shells**
2. Type **chsh** and press Enter key
3. You need to enter the new shell full path. For example, /bin/ksh
4. Log in and log out to verify that your shell changed corretly on Linux operating systems.

Let us see all commands in details.

## **List your shells in Linux**

Run the following [cat command](https://www.cyberciti.biz/faq/linux-unix-appleosx-bsd-cat-command-examples/) on the **/etc/shells** file:  
cat /etc/shells

**The shell’s treatment of the command-line**

The sequence in which the shell processes the command-line in one or more passes.

1. Parsing - The shell first breaks up the command line into words, using spaces and tabs as delimiters unless quoted. All consecutive occurrences of space or tab replaced with single space.
2. Variable evaluation – All words preceded by a $ are evaluated as variables, unless quoted or escaped

$x=10

$echo $x

1. Command substitution – Any command surrounded by back quotes is executed by the shell, which then replaces the standard output of the command into command line.

$echo “There are `ls|wc -w` files in the current directory

1. Redirection – The shell then looks for the characters >, < and >> to open the files that they point to.

$ ls > files.txt

1. Wild card interpretation – The shell finally scans the command line for wild cards \*, ?, [ ]. Any word containing a wild card is replaced by a sorted list of filenames that match the pattern and then forms as arguments to the command.

$ ls -l l\*

1. Path evaluation – It finally looks for path variable to determine the sequence of directories it has to search in order to hunt for the command.

$echo $PATH

The command is then passed on to the kernel for execution and the shell waits for its completion

**What is a built-in command in Linux?**

A builtin command is a Linux/Unix command which is "**built into a shell interpreter such as sh, ksh, bash, dash, csh etc**". Thats where the name came from for these built-in commands. In other words we can say that these commands will always available in RAM so that accessing them is bit fast when compared to external commands which are stored on hard disk. There are many advantages for this type of commands. Below are some advantages.

**Increase performance**

When we keep frequently execute commands in RAM they will decrease time delay in accessing them from disks and improve performance. Built-in commands will solve this problem by loading them self into RAM when an Interpreter is loaded in to

**Maintain set of commands at minimal shell**

Sometimes there is high chance of system crash and you cannot access anything on a machine. But if we still have shell access, we can execute all built-in commands to do basic troubleshooting and recovery system.

A *builtin* is a **command** contained within the Bash tool set, literally *built in*. This is either for performance reasons -- builtins execute faster than external commands, which usually require *forking off* a separate process -- or because a particular builtin needs direct access to the shell internals.

**How to list all built-in commands?**

There are many ways we can list builtin commands. Commands like compgen and help will help us to list all available builtin commands for a particular shell.  
Example:

**[surendra@linuxnix.com ~]$ compgen -b**

**Output:**

**.**

**:**

**[**

**alias**

**bg**

**bind**

**break**

**builtin**

**caller**

**cd**

**command**

**compgen**

**complete**

**compopt**

**….**

We should make note that**"builtin commands varies from shell to shell"**.

**How to check if a Linux command is built-in or not?**

To check if a command is external or internal command(built-in) we have to execute type command  
Example:

**[surendra@linuxnix.com ~]$ type cd**

Output:

**cd is a shell builtin**