

UNIT 3

Introduction to Linux

Linux and UNIX both are operating systems written in C and Assembly languages. UNIX is a primary operating system introduced in the year 1970 by Ken Thompson and others, while it is a multitasking and multiuser operating system.

Linux is an operating system, a software program that controls your computer. Most PC vendors load an operating system—generally, Microsoft Windows—onto the hard drive of a PC before delivering the PC; so, unless the hard drive of your PC has failed or you've upgraded your operating system, you may not understand the function of an operating system.

Like Unix, Linux can be generally divided into three major component of Linux OS :

- ❑ **Kernel**
 - **Manage hardware devices**
- ❑ **Environment**
 - **An Interface for the user**
- ❑ **File Structure**
 - **Organized the way files are stored on storage device**

The kernel is the core program that runs programs and manages hardware devices, such as disks and printers. The environment provides an interface for the user. It receives commands from the user and sends those commands to the kernel for execution. The file structure organizes the way files are stored on a storage device, such as a disk. Files are organized into directories. Each directory may contain any number of subdirectories, each holding files. Together, the kernel, the environment, and the file structure form the basic operating system structure. With these three, you can run programs, manage files, and interact with the system.

Suppose that, in a world without operating systems, you're programming a new PC application—perhaps a new multimedia word processor. Your application must cope with all the possible variations of PC hardware. As a result, it becomes bulky and complex. Users don't like it because it consumes too much hard drive space, takes a long time to load, and—because of its size and complexity—has more bugs than it should. Operating systems solve this problem by providing a standard way for applications to access hardware devices. Thanks to the operating system, applications can be more compact, because they share the commonly used code for accessing the hardware. Applications can also be more reliable, because common code is written only once—and by expert systems programmers rather than by application programmers.

The Evolution of Linux

Linux is an operating system for personal computers developed by Linus Torvalds at the university of Helsinki, Finland in 1991. Initially, Linux supported only the Intel 80x86 processor. Over the years, support has been added so that Linux can run on various other processors. Currently, Linux is one of very few operating systems that run on a wide range of processors, including Intel IA-32, Intel IA-64, AMD, DEC, PowerPC, Motorola, SPARC, and IBM S/390.

Linux is similar to UNIX in that it borrows many ideas from UNIX and implements the UNIX API. However, Linux is not a direct derivative of any particular UNIX distribution.

He published the Linux kernel under his own license and was restricted to use as commercially. Linux uses most of its tools from GNU software and are under GNU copyright. In 1992, he released the kernel under GNU General Public License.

Linux is undoubtedly the fastest-growing operating system today. It is used in areas such as embedded devices all the way to mainframes. One of the interesting and most important facts about Linux is that it is open-sourced. The Linux kernel is licensed under the GNU General Public License (GPL); the kernel source code is freely available and can be modified to suit the needs of your machine.

Linux Today

Today, supercomputers, smart phones, desktop, web servers, tablet, laptops and home appliances like washing machines, DVD players, routers, modems, cars, refrigerators, etc use Linux OS.

Difference between Linux and Unix

Linux

Linux is an open source multi-tasking, multi-user operating system. It was initially developed by Linus Torvalds in 1991. Linux OS is widely used in desktops, mobiles, mainframes etc.

Unix

Unix is multi-tasking, multi-user operating system but is not free to use and is not open source. It was developed in 1969 by Ken Thompson team at AT&T Bell Labs. It is widely used on servers, workstations etc. Following are the important differences between Linux and Unix.

Following are the important difference between Linux and Unix.

Sr. No.	Key	Linux	Unix
1	Development	Linux is open source and is developed by Linux community of developers.	Unix was developed by AT&T Bell labs and is not open source.
2	Cost	Linux is free to use.	Unix is licensed OS.

Sr. No.	Key	Linux	Unix
3	Supported File systems	Ext2, Ext3, Ext4, Jfs, ReiserFS, Xfs, Btrfs, FAT, FAT32, NTFS.	fs, gpfs, hfs, hfs+, ufs, xfs, zfs.
4	GUI	Linux uses KDE and Gnome. Other GUI supported are LXDE, Xfce, Unity, Mate.	Unix was initially a command based OS. Most of the unix distributions now have Gnome.
5	Usage	Linux is used in wide varieties from desktop, servers, smartphones to mainframes.	Unix is mostly used on servers, workstations or PCs.
6	Default Shell	Bash (Bourne Again SHell) is default shell for Linux.	Bourne Shell is default shell for Unix.
7	Target processor	Linux was initially developed for Intel's x86 hardware processors. Now it supports 20+ processor families.	CUnix supports PA-RISC and Itanium family.
8	Example	Ubuntu, Debian GNU, Arch Linux, etc.	SunOS, Solaris, SCO UNIX, AIX, HP/UX, ULTRIX etc.

Difference between Linux and Unix

Comparison	Linux	Unix
Definition	It is an open-source operating system which is <i>freely available to everyone</i> .	It is an operating system which <i>can be only used by its copyrighters</i> .
Examples	It has different distros like Ubuntu, Redhat, Fedora, etc	IBM AIX, HP-UX and Sun Solaris.
Users	Nowadays, Linux is in great demand. Anyone can use Linux whether a home user, developer or a student.	It was developed mainly for servers, workstations and mainframes.
Usage	Linux is used everywhere from servers, PC, smartphones, tablets to mainframes and supercomputers.	It is used in servers, workstations and PCs.

Cost	Linux is freely distributed,downloaded, and distributed through magazines also. And priced distros of Linux are also cheaper than Windows.	Unix copyright vendors decide different costs for their respective Unix Operating systems.
Development	As it is open source, it is developed by sharing and collaboration of codes by world-wide developers.	Unix was developed by AT&T Labs, various commercial vendors and non-profit organizations.
Manufacturer	Linux kernel is developed by the community of developers from different parts of the world. Although the father of Linux, Linus Torvalds oversees things.	Unix has three distributions IBM AIX, HP-UX and Sun Solaris. Apple also uses Unix to make OSX operating system.
GUI	Linux is command based but some distros provide GUI based Linux. Gnome and KDE are mostly used GUI.	Initially it was command based OS, but later Common Desktop Environment was created. Most Unix distributions use Gnome.
Interface	The default interface is BASH (Bourne Again SHell). But some distros have developed their own interfaces.	It originally used Bourne shell. But is also compatible with other GUIs.
File system support	Linux supports more file system than Unix.	It also supports file system but lesser than Linux.
Coding	Linux is a Unix clone,behaves like Unix but doesn't contain its code.	Unix contain a completely different coding developed by AT&T Labs.
Operating system	Linux is just the kernel.	Unix is a complete package of Operating system.
Security	It provides higher security. Linux has about 60-100 viruses listed till date.	Unix is also highly secured. It has about 85-120 viruses listed till date
Error detection and solution	As Linux is open-source,whenever a user post any kind of threat, developers from all over the world start working on it. And hence, it provides faster solution.	In Unix, users have to wait for some time for the problem to be resolved.

Technical characteristics of Linux :

- Cross-platform
- Cost
- Power
- Availability
- Reliability
- **Cross Platform**

Linux is a cross-platform operating system that runs on many computer models. Only Unix, an ancestor of Linux, rivals Linux in this respect. In comparison, Windows 2000 and XP run only on CPUs having the Intel-compatible architecture.

Cost

Foremost in the minds of many is the low cost of Linux. Comparable server operating systems can cost more than \$100,000. On the other hand, the low cost of Linux makes it practical for use even as a desktop operating system. In that mode, it truly eclipses the competition.

Power

Many desktop systems are employed as servers. Because of its design and heritage, the features and performance of Linux readily outshine those of desktop operating systems used as makeshift servers. For example, Microsoft's software license for Windows NT/2000/XP restricts the number of authenticated client connections; if you want your Windows NT/2000/XP server to be able to handle 100 authenticated clients, you must pay Microsoft a hefty license fee. However, Linux imposes no such restriction; your Linux desktop or server system is free to accept as many client connections as you think it can handle.

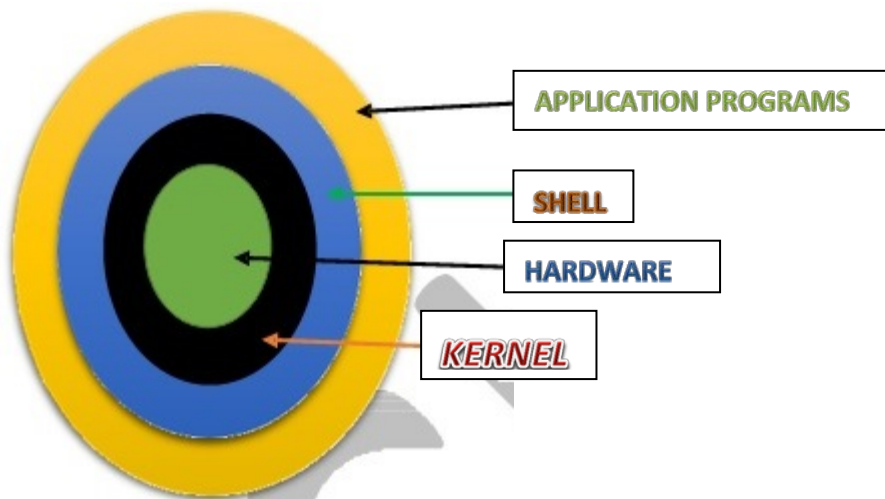
Linux distribution: a package that includes all the softwares needed to install and run Linux

❑ Popular Distribution :

- Debian
- Fedora Core
- Red Hat Enterprise Linux

Architecture of Linux operating system

The architecture of Linux is composed of kernel, shell and application programs that is softwares.



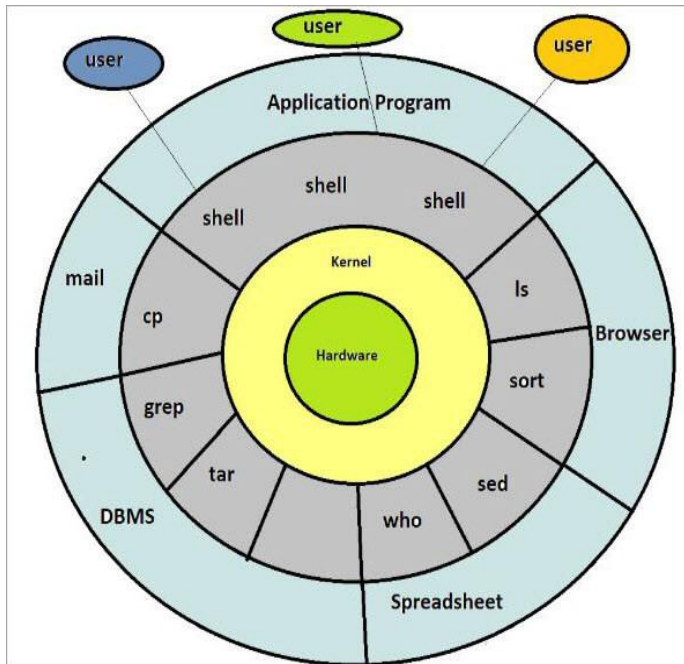
HARDWARE: physical parts of a computer, such as central processing unit (CPU), monitor, mouse, keyboard, hard disk and other connected devices to CPU.

KERNEL: A kernel is a computer program and is the central, core part of an operating system. It manages the operations of the computer and the hardware, most notably memory and CPU time. It is an integral part of any operating system.

SHELL: Shell is an environment in which we can run our commands, programs, and shell scripts. It is a user interface for access to an operating system's services. (User interface program execution, file system manipulation, input/output operations, communication, resource allocation, error detection, security and protection)

The diagram of kernel shell user relationship is as below:

THE KERNEL



1. Kernel is core (main) part of Linux operating system.
2. It is collection of routine communicate with hardware directly.
3. It loads into memory when Linux is booted.
4. Kernel provides support to user programs through system call.
5. Kernel manages Computer memory, schedules processes, decides priorities of processes and performs other tasks.
6. Kernel does lot of work even if no application software is running.
7. Hence kernel often called as application software gateway to the computer resources.
8. Kernel is represented by /boot/vmlinuz.

Shell

1. Shell is interface between user and kernel.
2. It is outer part of operating system.
3. A shell is a user interface for access to an operating system's services Shell is an environment in which we can run our commands, programs, softwares and shell scripts.
4. Computers do not have any inherent (मूळची) capability (क्षमता) of translating commands into actions, it is done by Shell.
5. There can be many shells in action - one shell for each user who logged in.

How Shell works?

When we enter commands through the keyboard, it gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. OR the shell thoroughly examines the keyboard input for special characters. If it finds any, it rebuilds a simplified command-line, and finally communicate with the Kernel to see that the command is executed.

To know the running shell, use echo \$SHELL command in terminal.

Application programs/software

An application, or application program, is a software program that runs on your computer. It is excited by user. Some inbuilt application programs in Linux are terminal, Firefox browser, Libre office

System calls

There are over a thousand commands available in Linux operating system. These all commands use a function to communicate with kernel - and it is called as system call. System call is the interface between a process and an operating system or System calls are the only entry points into the kernel system.

Ex. 1) a typical Linux writes a file with write system call. Same system call can access both a file and device.

2) Open system call opens both file and device. These system calls are built into kernel. And interaction through the system calls represents and efficient means of communication with operating system.

Linux Licensing

The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation

1 Not **public domain**, in that not all rights are waived

Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product

1 Can sell distributions, but must offer the source code too

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools

- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
 - Supports Pthreads and a subset of POSIX real-time process control
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System

nagement programs	user processes	user utility programs	compilers
system shared libraries			
Linux kernel			
loadable kernel modules			

The Linux Virtual File System

The Linux kernel implements the concept of Virtual File System (VFS, originally Virtual Filesystem Switch), so that it is (to a large degree) possible to separate actual "low-level" filesystem code from the rest of the kernel. The API of a filesystem is described below.

This API was designed with things closely related to the ext2 filesystem in mind. For very different filesystems, like NFS, there are all kinds of problems.

Four main objects: superblock, dentries, inodes, files

The kernel keeps track of files using in-core inodes ("index nodes"), usually derived by the low-level filesystem from on-disk inodes.

A file may have several names, and there is a layer of dentries ("directory entries") that represent pathnames, speeding up the lookup operation.

Several processes may have the same file open for reading or writing, and file structures contain the required information such as the current file position.

Access to a filesystem starts by mounting it. This operation takes a filesystem type (like ext2, vfat, iso9660, nfs) and a device and produces the in-core superblock that contains the information required for operations on the filesystem; a third ingredient, the mount point, specifies what pathname refers to the root of the filesystem.

Auxiliary objects

We have filesystem types, used to connect the name of the filesystem to the routines for setting it up (at mount time) or tearing it down (at umount time).

A struct `vfsmount` represents a subtree in the big file hierarchy - basically a pair (device, mountpoint).

A struct `nameidata` represents the result of a lookup.

A struct `address_space` gives the mapping between the blocks in a file and blocks on disk. It is needed for I/O.

Terminology

Various objects play a role here. There are file systems, organized collections of files, usually on some disk partition. And there are filesystem types, abstract descriptions of the way data is organized in a filesystem of that type, like FAT16 or ext2. And there is code, perhaps a module, that implements the handling of file systems of a given type. Sometimes this code is called a low-level filesystem, low-level since it sits below the VFS just like low-level SCSI drivers sit below the higher SCSI layers.

Filesystem type registration

A module implementing a filesystem type must announce its presence so that it can be used. Its task is (i) to have a name, (ii) to know how it is mounted, (iii) to know how to lookup files, (iv) to know how to find (read, write) file contents.

This announcing is done using the call `register_filesystem()`, either at kernel initialization time or when the module is inserted. There is a single argument, a struct that contains the name of the filesystem type (so that the kernel knows when to invoke it) and a routine that can produce a superblock.

The struct is of type `struct file_system_type`. Here the 2.2.17 version:

Struct `file_system_type`

```
struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*get_sb)(struct file_system_type *,
        int, char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type *next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
};
```

Let us look at the fields of the struct `file_system_type`.

name

Here the filesystem type gives its name ("tue"), so that the kernel can find it when someone does `mount -t tue /dev/foo /dir`. (The name is the third parameter of the mount system call.) It

must be non-NULL. The name string lives in module space. Access must be protected either by a reference to the module, or by the `file_systems_lock`.

get_sb

At mount time the kernel calls the `fstype->get_sb()` routine that initializes things and sets up a superblock. It must be non-NULL. Typically this is a 1-line routine that calls one of `get_sb_bdev`, `get_sb_single`, `get_sb_nodev`, `get_sb_pseudo`.

The routines `get_sb_single` and `get_sb_nodev` are almost identical. Both are for virtual filesystems. The former is used when there can be at most one instance of the filesystem. (Now an old instance is used if there is one, but its flags may be changed.)

kill_sb

At umount time the kernel calls the `fstype->kill_sb()` routine to clean up. It must be non-NULL. Typically one of `kill_block_super`, `kill_anon_super`, `kill_litter_super`.

The first is normal for filesystems backed by block devices. The second for virtual filesystems, where the information is generated on the fly. The third for in-memory filesystems without backing store - they need an additional `dget()` when a file is created (so that their dentries always have a nonzero reference count and are not garbage collected), and the `d_genocide()` that is the difference between `kill_anon_super` and `kill_litter_super` does the balancing `dput()`.

fs_flags

The `fs_flags` field of a struct `file_system_type` is a bitmap, an OR of several possible flags with mostly obscure uses only. The flags are defined in `fs.h`. This field was introduced in 2.1.43. The number of flags, and their meanings, varies. In 2.6.19 there are the four flags `FS_REQUIRES_DEV`, `FS_BINARY_MOUNTDATA`, `FS_REVAL_DOT`, `FS_RENAME_DOES_D_MOVE`.

- `FS_REQUIRES_DEV`

The `FS_REQUIRES_DEV` flag (since 2.1.43) says that this is not a virtual filesystem - an actual underlying block device is required. It is used in only two places: when `/proc/filesystems` is generated, its absence causes the filesystem type name to be prefixed by "nodev". And in `fs/nfsd/export.c` this flag is tested in the process of determining whether the filesystem can be exported via NFS. Earlier there were more uses.

- `FS_BINARY_MOUNTDATA`

The `FS_BINARY_MOUNTDATA` flag (since 2.6.5) is set to tell the selinux code that the mount data is binary, and cannot be handled by the standard option parser. (This flag is set for `afs`, `coda`, `nfs`, `smbfs`.)

- `FS_REVAL_DOT`

The `FS_REVAL_DOT` flag (since 2.6.0test4) is set to tell the VFS code (in `namei.c`) to revalidate the paths `"/"`, `"/."`, `"/.."` since they might have gone stale. (This flag is set for NFS.)

- `FS_RENAME_DOES_D_MOVE`

The `FS_RENAME_DOES_D_MOVE` flag (since 2.6.19) says that the low-level filesystem will handle `d_move()` during a `rename()`. Earlier (2.4.0test6-2.6.19) this was called `FS_ODD_RENAME` and was used for NFS only, but now this is also useful for ocfs2. See also the discussion of [silly rename](#).

- `FS_NOMOUNT` (gone)

The `FS_NOMOUNT` flag (2.3.99pre7-2.5.22) says that this filesystem must never be mounted from userland, but is used only kernel-internally. This was used, for example, for pipefs, the implementation of Unix pipes using a kernel-internal filesystem (see `fs/pipe.c`). Even though the flag has disappeared, the concept remains, and is now represented by the `MS_NOUSER` flag.

- `FS_LITTER` (gone)

The `FS_LITTER` flag (2.4.0test3-2.5.7) says that after `umount` a `d_genocide()` is needed. This will remove one reference from all dentries in that tree, probably killing all of them, which is necessary in case at creation time the dentries already got reference count 1. (This is typically done for an in-core filesystem where dentries cannot be recreated when needed.) This flag disappeared in Linux 2.5.7 when the explicit `kill_super` method `kill_litter_super` was introduced.

- `FS_SINGLE` (gone)

The `FS_SINGLE` flag (2.3.99pre7-2.5.4) says that there is only a single superblock for this filesystem type, so that only a single instance of this filesystem may exist, possibly mounted in several places.

- `FS_IBASKET` and `FS_NO_DCACHE` and `FS_NO_PRELIM` (gone)

The `FS_IBASKET` was defined in 2.1.43 but never used, and the definition disappeared in 2.3.99pre4. The `FS_NO_DCACHE` and `FS_NO_PRELIM` flags were introduced in 2.1.43, but were a mistake and disappeared again in 2.1.44. However, the definitions survived until Linux 2.5.22. For the purposes of these flags, see the comment in 2.1.43:dcache.c.

owner

The `owner` field of a struct `file_system_type` points at the module that owns this struct. When doing things that might sleep, we must make sure that the module is not unloaded while we are using its data, and do this with `try_inc_mod_count(owner)`. If this fails then the module was just unloaded. If it succeeds we have incremented a reference count so that the module will not go away before we are done.

This field is `NULL` for filesystems compiled into the kernel.

Mounting

The mount system call attaches a filesystem to the big file hierarchy at some indicated point. Ingredients needed: (i) a device that carries the filesystem (disk, partition, floppy, CDROM, SmartMedia card, ...), (ii) a directory where the filesystem on that device must be attached, (iii) a filesystem type.

In many cases it is possible to guess (iii) given the bits on the device, but heuristics fail in rare cases. Moreover, sometimes there is no difference on the device, as for example in the case where a FAT filesystem without long filenames must be mounted. Is it msdos? or vfat? That information is only in the user's head. If it must be used later in an environment that cannot handle long filenames it should be mounted as msdos; if files with long names are going to be copied to it, as vfat.

The kernel does not guess (except perhaps at boot time, when the root device has to be found), and requires the three ingredients. In fact the mount system call has five parameters: there are also mount flags (like "read-only") and options, like for ext2 the choice between errors=continue and errors=remount-ro and errors=panic.

The superblock

The superblock gives global information on a filesystem: the device on which it lives, its block size, its type, the dentry of the root of the filesystem, the methods it has, etc., etc.

```
struct super_block {
    dev_t s_dev;
    unsigned long s_blocksize;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dentry *s_root;
    ...
}
```

Each superblock is on six lists, with links through the fields s_list, s_dirty, s_io, s_anon, s_files, s_instances, respectively.

The super_blocks list

All superblocks are collected in a list super_blocks with links in the fields s_list. This list is protected by the spinlock sb_lock. The main use is in super.c:get_super() or user_get_super() to find the superblock for a given block device. (Both routines are identical, except that one takes a bdev, the other a dev_t.) This list is also used various places where all superblocks must be sync'ed or all dirty inodes must be written out.

The fs_supers list

All superblocks of a given type are collected in a list headed by the fs_supers field of the struct file_system_type, with links in the fields s_instances. Also this list is protected by the spinlock sb_lock. See above.

The file list

All open files belonging to a given superblock are chained in a list headed by the s_files field of the superblock, with links in the fields f_list of the files. These lists are protected by the spinlock files_lock. This list is used for example in fs_may_remount_ro() to check that there are no files currently open for writing. See also below.

The list of anonymous dentries

Normally, all dentries are connected to root. However, when NFS filehandles are used this need not be the case. Dentries that are roots of subtrees potentially unconnected to root are chained in a list headed by the s_anon field of the superblock, with links in the fields d_hash. These lists are

protected by the spinlock `dcache_lock`. They are grown in `dcache.c:d_alloc_anon()` and shrunk in `super.c:generic_shutdown_super()`. See the discussion in Documentation/filesystems/Exporting.

The inode lists `s_dirty`, `s_io`

Lists of inodes to be written out. These lists are headed at the `s_dirty` (resp. `s_io`) field of the superblock, with links in the fields `i_list`. These lists are protected by the spinlock `inode_lock`. See `fs/fs-writeback.c`.

Inodes

An (in-core) inode contains the metadata of a file: its serial number, its protection (mode), its owner, its size, the dates of last access, creation and last modification, etc. It also points to the superblock of the filesystem the file is in, the methods for this file, and the dentries (names) for this file.

```
struct inode {
    unsigned long i_ino;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    kdev_t i_rdev;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_ctime;
    struct timespec i_mtime;
    struct super_block *i_sb;
    struct inode_operations *i_op;
    struct address_space *i_mapping;
    struct list_head i_dentry;
    ...
}
```

Each inode is on four lists, with links through the fields `i_hash`, `i_list`, `i_dentry`, `i_devices`.

The dentry list

All dentries belonging to this inode (names for this file) are collected in a list headed by the inode field `i_dentry` with links in the dentry fields `d_alias`. This list is protected by the spinlock `dcache_lock`.

The hash list

All inodes live in a hash table, with hash collision chains through the field `i_hash` of the inode. These lists are protected by the spinlock `inode_lock`. The appropriate head is found by a hash function; it will be an element of the `inode_hashtable[]` array when the inode belongs to a superblock, or `anon_hash_chain` if not.

`i_list`

Inodes are collected into lists that use the `i_list` field as link field. The lists are protected by the spinlock `inode_lock`. An inode is either unused, and then on the chain with head `inode_unused`, or in use but not dirty, and then on the chain with head `inode_in_use`, or dirty, and then on one of the per-superblock lists with heads `s_dirty` or `s_io`, see above.

`i_devices`

Inodes belonging to a given block device are collected into a list headed by the `bd_inodes` field of the block device, with links in the inode `i_devices` fields. The list is protected by

the `bdev_lock` spinlock. It is used to set the `i_bdev` field to `NULL` and to reset `i_mapping` when the block device goes away.

Dentries

The dentries encode the filesystem tree structure, the names of the files. Thus, the main parts of a dentry are the inode (if any) that belongs to it, the name (the final part of the pathname), and the parent (the name of the containing directory). There are also the superblocks, the methods, a list of subdirectories, etc.

```
struct dentry {
    struct inode *d_inode;
    struct dentry *d_parent;
    struct qstr d_name;
    struct super_block *d_sb;
    struct dentry_operations *d_op;
    struct list_head d_subdirs;
    ...
}
```

Each dentry is on five lists, with links through the fields `d_hash`, `d_lru`, `d_child`, `d_subdirs`, `d_alias`.

Naming flaw

Some of these names were badly chosen, and lead to confusion. We should do a global replace changing `d_subdirs` into `d_children` and `d_child` into `d_sibling`.

Value of a dentry

The pathname represented by a dentry, is the concatenation of the name of its parent `d_parent`, a slash character, and its own name `d_name`.

However, if the dentry is the root of a mounted filesystem (i.e., if `dentry->d_covers != dentry`), then its pathname is the pathname of the mount point `d_covers`. Finally, the pathname of the root of the filesystem (with `dentry->d_parent == dentry`) is `"/"`, and this is also its `d_name`.

The `d_mounts` and `d_covers` fields of a dentry point back to the dentry itself, except that the `d_covers` field of the dentry for the root of a mounted filesystem points back to the dentry for the mount point, while the `d_mounts` field of the dentry for the mount point points at the dentry for the root of a mounted filesystem.

The `d_parent` field of a dentry points back to the dentry for the directory in which it lives. It points back to the dentry itself in case of the root of a filesystem.

A dentry is called negative if it does not have an associated inode, i.e., if it is a name only.

We see that although a dentry represents a pathname, there may be several dentries for the same pathname, namely when overmounting has taken place. Such dentries have different inodes.

Of course the converse, an inode with several dentries, can also occur.

The above description, with `d_mounts` and `d_covers`, was for 2.4. In 2.5 these fields have disappeared, and we only have the integer `d_mounted` that indicates how many filesystems have been mounted at that point. In case it is nonzero (this is what `d_mountpoint()` tests), a hash table lookup can find the actual mounted filesystem.

d_hash

Dentries are used to speed up the lookup operation. A hash table `dentry_hashtable` is used, with an index that is a hash of the name and the parent. The hash collision chain has links through the dentry fields `d_hash`. This chain is protected by the spinlock `dcache_lock`.

d_lru

All unused dentries are collected in a list `dentry_unused` with links in the dentry fields `d_lru`. This list is protected by the spinlock `dcache_lock`.

d_child, d_subdirs

All subdirectories of a given directory are collected in a list headed by the dentry field d_subdirs with links in the dentry fields d_child. These lists are protected by the spinlock dcache_lock.

d_alias

All dentries belonging to the same inode are collected in a list headed by the inode field i_dentry with links in the dentry fields d_alias. This list is protected by the spinlock dcache_lock.

Essential Linux Commands

The LINUX system is command-based i.e things happen because of the commands that you key in. All LINUX commands are seldom more than four characters long.

They are grouped into two categories:

- **Internal Commands :** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable, and also no process needs to be spawned for executing it.
Examples: source, cd, fg, etc.
- **External Commands :** Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable, and also a new process has to be spawned and the command gets executed. They are usually located in /bin or /usr/bin. For example, when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed.
Examples: ls, cat etc.

the ls command. Since ls is a program or file having an independent existence in the /bin directory(or /usr/bin), it is branded as an external command that actually means that the ls command is not built into the shell and these are executables present in a separate file. In simple words, when you will key in the ls command, to be executed it will be found in /bin. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by PATH. For instance, take echo command:

```
$type echo
```

echo is a shell builtin

echo isn't an external command in the sense that, when you type echo, the shell won't look in its PATH to locate it(even if it is there in /bin). Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which echo is a member, are known as internal commands.

Getting the list of Internal Commands

If you are using bash shell you can get the list of shell built-in commands with help command :
\$help

```
// this will list all
the shell built-in commands //
```

How to find out whether a command is internal or external?

In addition to this you can also find out about a particular command i.e whether it is internal or external with the help of type command :

```
$type cat
cat is /bin/cat
```

```
//specifying that cat is
external type//
```

```
$type cd
cd is a shell builtin
```

```
//specifying that cd is
internal type//
```

Internal vs External

The question that when to use which command between internal and external command is of no use cause the user uses a command according to the need of the problem he wants to solve. The only difference that exists between internal and external commands is that internal commands work much faster than the external ones as the shell has to look for the path when it comes to the use of external commands.

There are some cases where you can avoid the use of external by using internal in place of them, like if you need to add two numbers you can do it as:

```
//use of internal command let
```

```
for addition//
```

```
$let c=a+b
instead of using :
```

```
//use of external command expr
```

```
for addition//
```

```
$c=`expr $a+$b`
```

In such a case, the use of let will be a better option as it is a shell built-in command so it will work faster than the expr which is an external command.

Kernel Process Management in Linux

The *process* is one of the fundamental abstractions in Unix operating systems¹. A process is a program (object code stored on some media) in execution. Processes are, however, more than just the executing program code (often called the *text section* in Unix). They also include a set of resources such as open files and pending signals, internal kernel data, processor state, an address space, one or more *threads of execution*, and a *data section* containing global variables. Processes, in effect, are the living result of running program code.

Threads of execution, often shortened to *threads*, are the objects of activity within the process. Each thread includes a unique program counter, process stack, and set of processor registers. The kernel schedules individual threads, not processes. In traditional Unix systems, each process consists of one thread. In modern systems, however, multithreaded programs—those that consist of more than one thread—are common. As you will see later, Linux has a unique implementation of threads: It does not differentiate between threads and processes. To Linux, a thread is just a special kind of process.

On modern operating systems, processes provide two virtualizations: a virtualized processor and virtual memory. The virtual processor gives the process the illusion that it alone monopolizes the system, despite possibly sharing the processor among dozens of other processes. Chapter 4, "Process Scheduling," discusses this virtualization. Virtual memory lets the process allocate and manage memory as if it alone owned all the memory in the system. Virtual memory is covered in Chapter 11, "Memory Management." Interestingly, note that threads *share* the virtual memory abstraction while each receives its own virtualized processor.

A program itself is not a process; a process is an *active* program and related resources. Indeed, two or more processes can exist that are executing the *same* program. In fact, two or more processes can exist that share various resources, such as open files or an address space.

A process begins its life when, not surprisingly, it is created. In Linux, this occurs by means of the `fork()` system call, which creates a new process by duplicating an existing one. The process that calls `fork()` is the *parent*, whereas the new process is the *child*. The parent resumes execution and the child starts execution at the same place, where the call returns. The `fork()` system call returns from the kernel twice: once in the parent process and again in the newborn child.

Often, immediately after a `fork` it is desirable to execute a new, different, program. The `exec*()` family of function calls is used to create a new address space and load a new program into it. In modern Linux kernels, `fork()` is actually implemented via the `clone()` system call, which is discussed in a following section.

Finally, a program exits via the `exit()` system call. This function terminates the process and frees all its resources. A parent process can inquire about the status of a terminated child via the `wait4()`² system call, which enables a process to wait for the termination of a specific process. When a process exits, it is placed into a special zombie state that is used to represent terminated processes until the parent calls `wait()` or `waitpid()`.

Another name for a process is a *task*. The Linux kernel internally refers to processes as tasks. In this book, I will use the terms interchangeably, although when I say *task* I am generally referring to a process from the kernel's point of view.

Process Descriptor and the Task Structure

The `task_struct` is a relatively large data structure, at around 1.7 kilobytes on a 32-bit machine. This size, however, is quite small considering that the structure contains all the information that the kernel has and needs about a process. The process descriptor contains the data that describes the executing program—open files, the process's address space, pending signals, the process's state, and much more

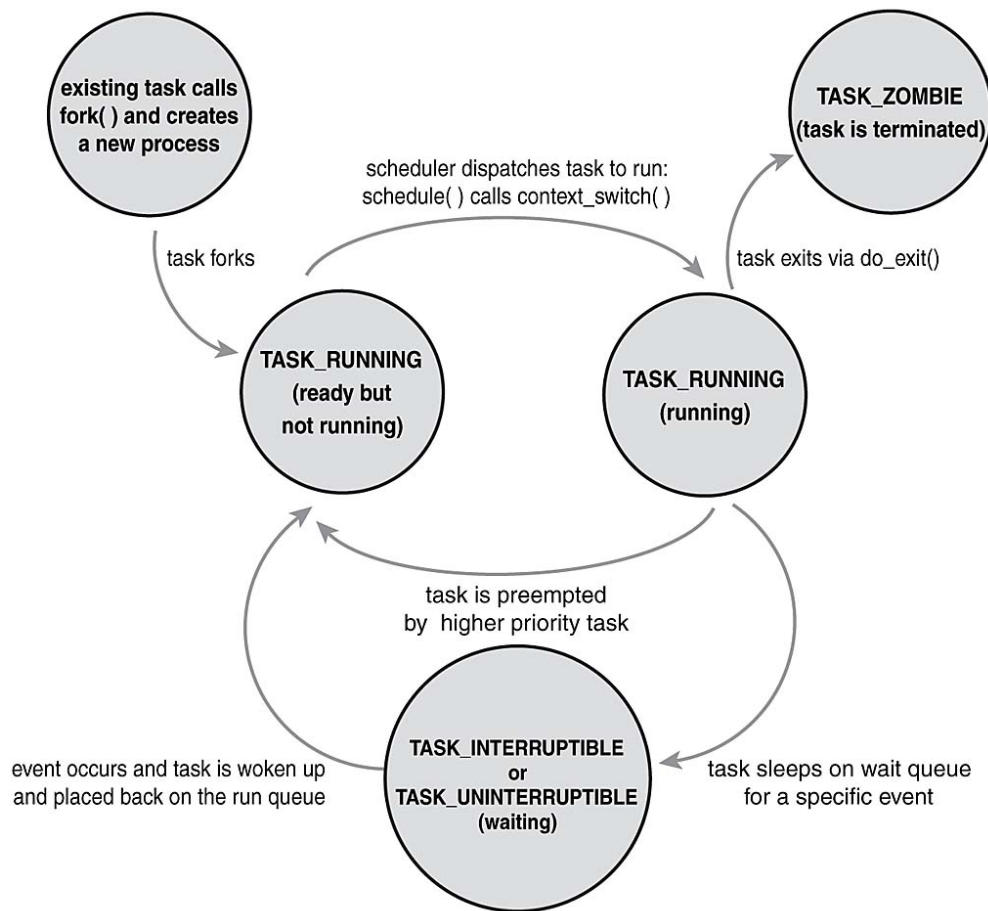
Allocating the Process Descriptor

The `task_struct` structure is allocated via the *slab allocator* to provide object reuse and cache coloring. Prior to the 2.6 kernel series, `struct task_struct` was stored at the end of the kernel stack of each process. This allowed architectures with few registers, such as x86, to calculate the location of the process descriptor via the *stack pointer* without using an extra register to store the location. With the process descriptor now dynamically created via the slab allocator, a new structure, `struct thread_info`, was created that again lives at the bottom of the stack and at the top of the stack.

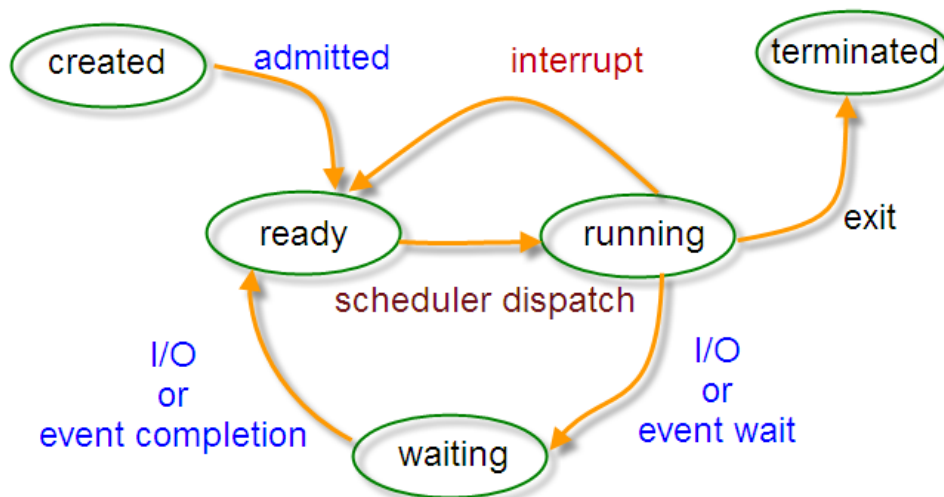
Process State

The state field of the process descriptor describes the current condition of the process. Each process on the system is in exactly one of five different states. This value is represented by one of five flags:

- **TASK_RUNNING**—The process is runnable; it is either currently running or on a runqueue waiting to run (runqueues are discussed in Chapter 4, "Scheduling"). This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.
- **TASK_INTERRUPTIBLE**—The process is sleeping (that is, it is blocked), waiting for some condition to exist. When this condition exists, the kernel sets the process's state to **TASK_RUNNING**. The process also awakes prematurely and becomes runnable if it receives a signal.
- **TASK_UNINTERRUPTIBLE**—This state is identical to **TASK_INTERRUPTIBLE** except that it does *not* wake up and become runnable if it receives a signal. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly. Because the task does not respond to signals in this state, **TASK_UNINTERRUPTIBLE** is less often used than **TASK_INTERRUPTIBLE**.
- **TASK_ZOMBIE**—The task has terminated, but its parent has not yet issued a `wait4()` system call. The task's process descriptor must remain in case the parent wants to access it. If the parent calls `wait4()`, the process descriptor is deallocated.
- **TASK_STOPPED**—Process execution has stopped; the task is not running nor is it eligible to run. This occurs if the task receives the **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signal or if it receives *any* signal while it is being debugged.



Process State



Linux - Signals

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals,

such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

The following table lists out common signals you might encounter and want to use in your programs –

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C)
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D)
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT _v	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)

SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

Signal is a notification, a message sent by either operating system or some application to our program. Signals are a mechanism for one-way asynchronous notifications. A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself. Signal typically alert a process to some event, such as a segmentation fault, or the user pressing Ctrl-C.

Linux kernel implements about 30 signals. Each signal identified by a number, from 1 to 31. Signals don't carry any argument and their names are mostly self explanatory. For instance SIGKILL or signal number 9 tells the program that someone tries to kill it, and SIGHUP used to signal that a terminal hangup has occurred, and it has a value of 1 on the i386 architecture.

With the exception of SIGKILL and SIGSTOP which always terminates the process or stops the process, respectively, processes may control what happens when they receive a signal. They can

1. accept the default action, which may be to terminate the process, terminate and coredump the process, stop the process, or do nothing, depending on the signal.
2. Or, processes can elect to explicitly ignore or handle signals.
 1. **Ignored signals** are silently dropped.
 2. **Handled signals** cause the execution of a user-supplied **signal handler** function. The program jumps to this function as soon as the signal is received, and the control of the program resumes at the previously interrupted instructions.

Default Actions

Every signal has a default action associated with it. The default action for a signal is the action that a script or program performs when it receives a signal.

Some of the possible default actions are –

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called **core** containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

Sending Signals

There are several methods of delivering signals to a program or script. One of the most common is for a user to type CONTROL-C or the INTERRUPT key while a script is executing.

When you press the Ctrl+C key, a SIGINT is sent to the script and as per defined default action script terminates.

Trapping Signals

When you press the Ctrl+C or Break key at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returns. This may not always be desirable. For instance, you may end up leaving a bunch of temporary files that won't get cleaned up.

Trapping these signals is quite easy, and the trap command has the following syntax –

```
$ trap commands signals
```

Here command can be any valid Unix command, or even a user-defined function, and signal can be a list of any number of signals you want to trap.

There are two common uses for trap in shell scripts –

- Clean up temporary files
- Ignore signals

UNIT – 4

Introduction to Linux Shell and Shell Scripting

If you are using any major operating system you are indirectly interacting to **shell**. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal. In this article I will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies –

- Kernel
- Shell
- Terminal

What is Kernel

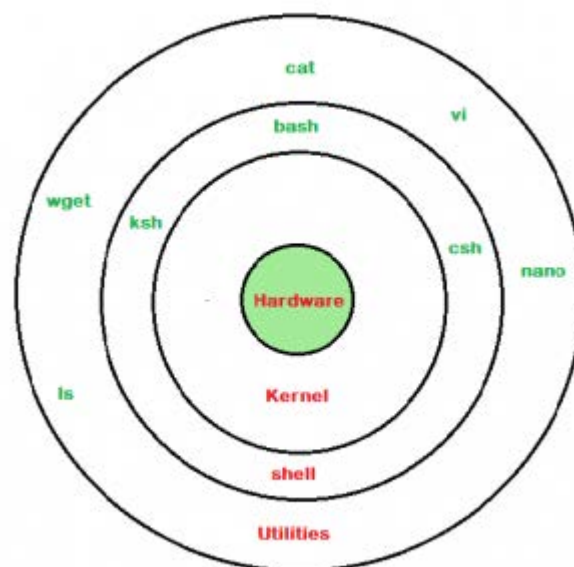
The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

- File management
- Process management
- I/O management
- Memory management
- Device management etc.

It is often mistaken that **Linus Torvalds** has developed Linux OS, but actually he is only responsible for development of Linux kernel. Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

What is Shell

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.



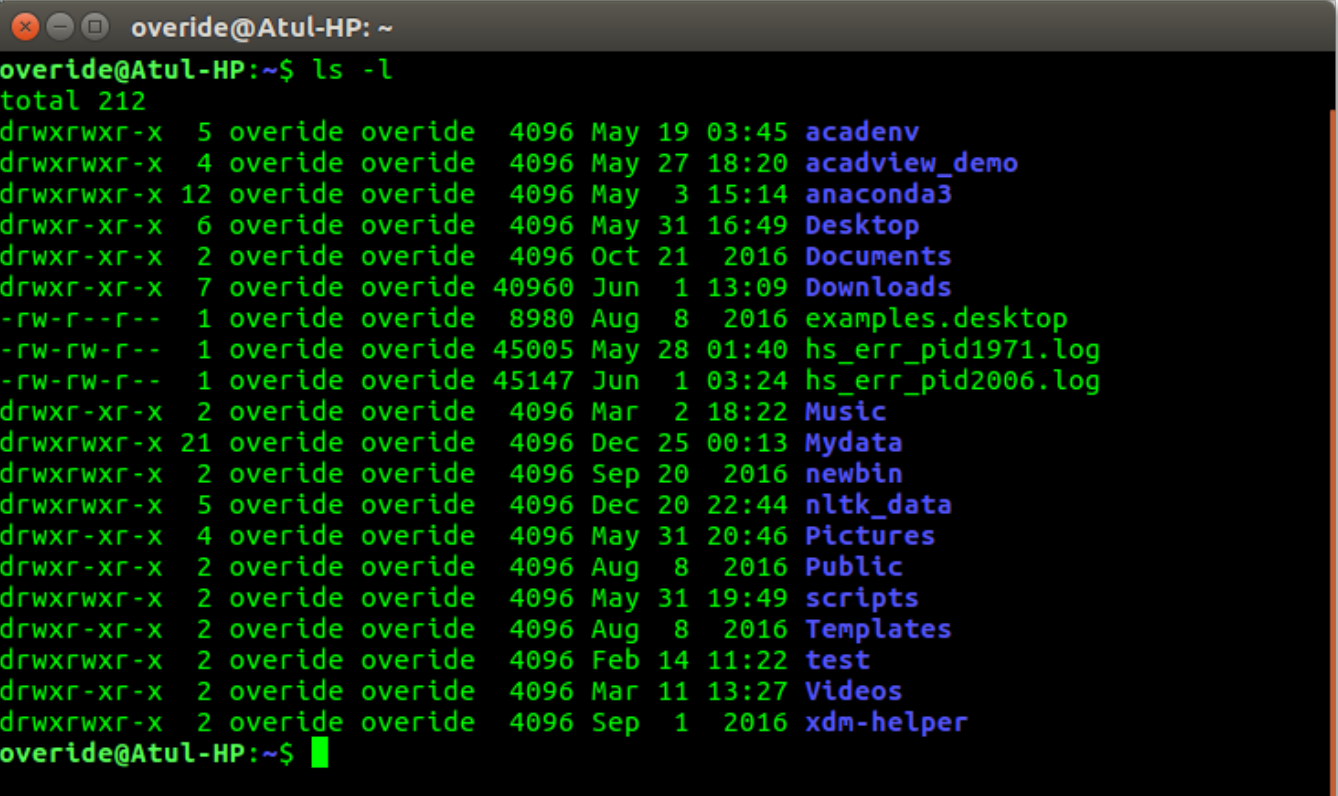
linux shell

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user. A terminal in Ubuntu 16.4 system looks like this –



```

override@Atul-HP: ~
override@Atul-HP:~$ ls -l
total 212
drwxrwxr-x  5 override override 4096 May 19 03:45 acadenv
drwxrwxr-x  4 override override 4096 May 27 18:20 acadview_demo
drwxrwxr-x 12 override override 4096 May  3 15:14 anaconda3
drwxr-xr-x  6 override override 4096 May 31 16:49 Desktop
drwxr-xr-x  2 override override 4096 Oct 21  2016 Documents
drwxr-xr-x  7 override override 40960 Jun  1 13:09 Downloads
-rw-r--r--  1 override override 8980 Aug  8  2016 examples.desktop
-rw-rw-r--  1 override override 45005 May 28 01:40 hs_err_pid1971.log
-rw-rw-r--  1 override override 45147 Jun  1 03:24 hs_err_pid2006.log
drwxr-xr-x  2 override override 4096 Mar  2 18:22 Music
drwxrwxr-x 21 override override 4096 Dec 25 00:13 Mydata
drwxrwxr-x  2 override override 4096 Sep 20  2016 newbin
drwxrwxr-x  5 override override 4096 Dec 20 22:44 nltk_data
drwxr-xr-x  4 override override 4096 May 31 20:46 Pictures
drwxr-xr-x  2 override override 4096 Aug  8  2016 Public
drwxrwxr-x  2 override override 4096 May 31 19:49 scripts
drwxr-xr-x  2 override override 4096 Aug  8  2016 Templates
drwxrwxr-x  2 override override 4096 Feb 14 11:22 test
drwxr-xr-x  2 override override 4096 Mar 11 13:27 Videos
drwxrwxr-x  2 override override 4096 Sep  1  2016 xdm-helper
override@Atul-HP:~$ █

```

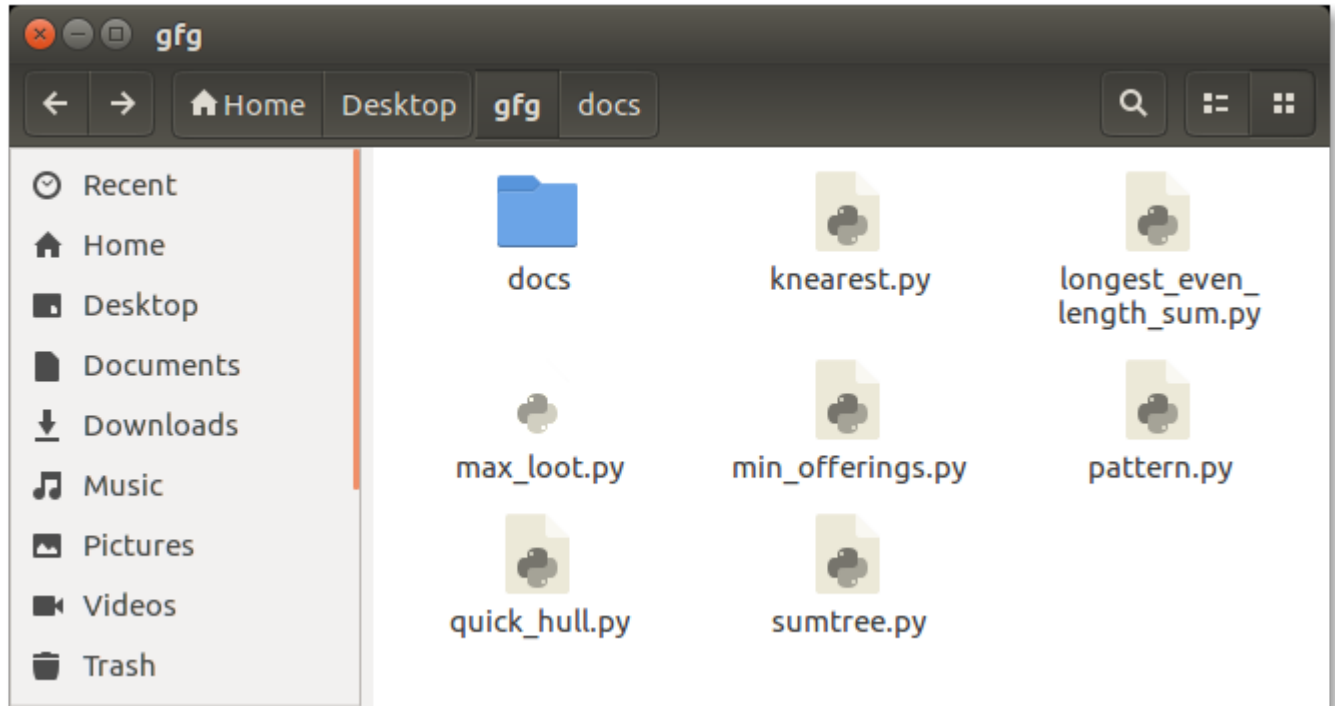
linux command line

In above screenshot “ls” command with “-l” option is executed.

It will list all the files in current working directory in long listing format. Working with command line shell is bit difficult for the beginners because it’s hard to memorize so many commands. It is very powerful, it allows user to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called **batch files** in Windows and **Shell Scripts** in Linux/macOS systems.

Graphical Shells

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions. A typical GUI in Ubuntu system –



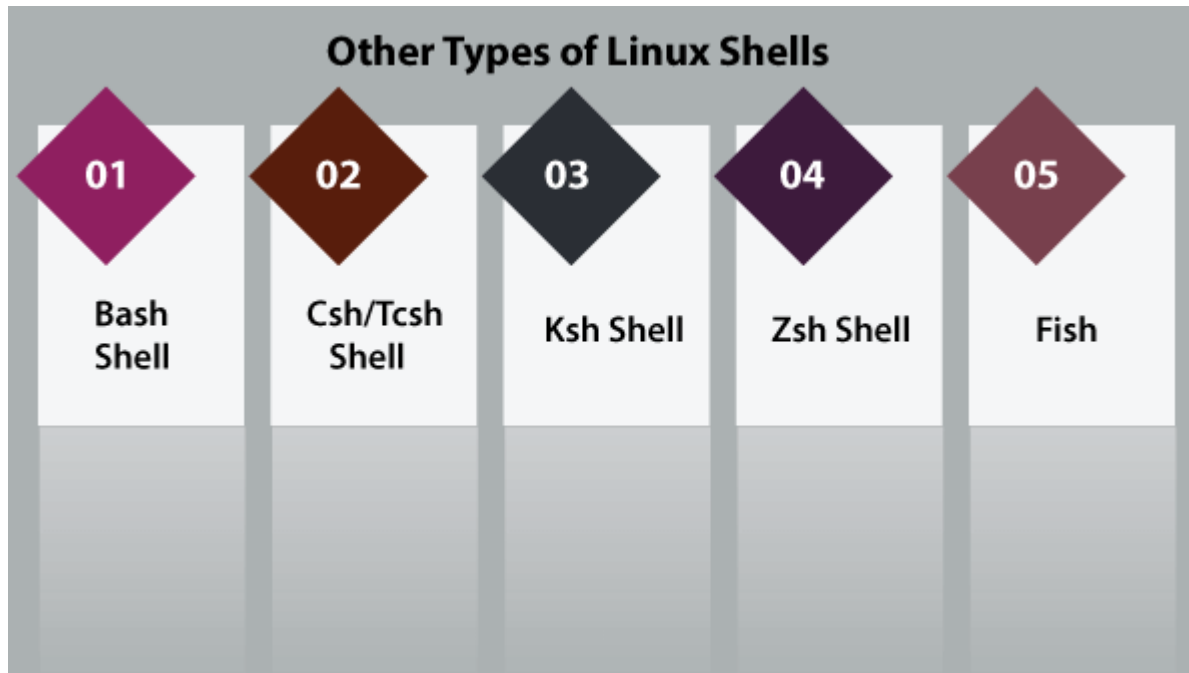
GUI shell

There are several shells are available for Linux systems like –

- BASH (Bourne Again SHell) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- CSH (C SHell) – The C shell's syntax and usage are very similar to the C programming language.
- KSH (Korn SHell) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built in functions.

There are various types of shells which are discussed as follows:



Bash Shell

In the bash shell, bash means Bourne Again Shell. It is a default shell over several distributions of Linux today. It is a sh-compatible shell. It could be installed over Windows OS. It facilitates practical improvements on sh for interactive and programming use which contains:

- Job Control
- Command-line editing
- Shell Aliases and Functions
- Unlimited size command history
- Integer arithmetic in a base from 2-64

Csh/Tcsh Shell

Tcsh is an upgraded C shell. This shell can be used as a shell script command processor and interactive login shell.

Tcsh shell includes the following characteristics:

- C like syntax
- Filename completion and programmable word
- Command-line editor
- Job control
- Spelling correction

Ksh Shell

Ksh means for **Korn shell**. It was developed and designed by **David G. Korn**. Ksh shell is a high-level, powerful, and complete programming language and it is a reciprocal command language as well just like various other GNU/Unix Linux shells. The usage and syntax of the C shell are very same as the C programming language.

Zsh Shell

Zsh shell is developed to be reciprocal and it combines various aspects of other GNU/Unix Linux shells like ksh, tcsh, and bash. Also, the POSIX shell standard specifications were based on the Korn shell. Also, it is a strong scripting language like other available shells. Some of its unique features are listed as follows:

- Startup files
- Filename generation
- Login/Logout watching
- Concept index
- Closing comments
- Variable index
- Key index
- Function index and various others that we could find out within the man pages.

All these shells do a similar job but take different commands and facilitate distinct built-in functions.

Fish

Fish stands for "**friendly interactive shell**". It was produced in 2005. Fish shell was developed to be fully user-friendly and interactive just like other shells. It contains some good features which are mentioned below:

- Web-based configuration
- Man page completions
- Auto-suggestions
- Support for term256 terminal automation
- Completely scripted with clean scripts

Shell Scripting

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal. As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell**

Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.
-

Why do we need shell scripts

There are many reasons to write shell scripts –

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. Etc

Simple demo of shell scripting using Bash Shell

If you work on terminal, something you traverse deep down in directories. Then for coming few directories up in path we have to execute command like this as shown below to get to the

“python”

directory

—

```

override@Atul-HP: ~/Mydata/project/python
override@Atul-HP:~/Mydata/project/python/projects/machine_learning/DSfromScratch/hypothe
sis$ cd ../../../../
override@Atul-HP:~/Mydata/project/python$ █

```

It is quite frustrating, so why not we can have a utility where we just have to type the name of directory and we can directly jump to that without executing “**cd ../**” command again and again. Save the script as “**jump.sh**”

For now we cannot execute our shell script because it do not have permissions. We have to make it executable by typing following command –

```
$ chmod -x path/to/our/file/jump.sh
```

Now to make this available on every terminal session, we have to put this in “**.bashrc**” file.

“**.bashrc**” is a shell script that Bash shell runs whenever it is started interactively. The purpose of a **.bashrc** file is to provide a place where you can set up variables, functions and aliases, define our prompt and define other settings that we want to use whenever we open a new terminal window.

Now open terminal and type following command –

```
$ echo "source ~/path/to/our/file/jump.sh">> ~/.bashrc
```

Now open you terminal and try out new “jump” functionality by typing following command-

```
$ jump dir_name
```

just like below screenshot –

```

override@Atul-HP: ~/Mydata/project/python
override@Atul-HP:~/Mydata/project/python/projects/machine_learning/DSfromScratch/hypothesis$ jump python
override@Atul-HP:~/Mydata/project/python$ █
  
```

I/O Redirection and Piping

One of the most important and interesting topics under Linux administration is I/O redirection. This feature of the command line enables you to redirect the input and/or output of commands from and/or to files, or join multiple commands together using pipes to form what is known as a “**command pipeline**”.

All the commands that we run fundamentally produce two kinds of output:

- the command result – data the program is designed to produce, and
- the program status and error messages that informs a user of the program execution details.

In Linux and other Unix-like systems, there are three default files named below which are also identified by the shell using file descriptor numbers:

- **stdin or 0** – it’s connected to the keyboard, most programs read input from this file.
- **stdout or 1** – it’s attached to the screen, and all programs send their results to this file and
- **stderr or 2** – programs send status/error messages to this file which is also attached to the screen.

Therefore, I/O redirection allows you to alter the input source of a command as well as where its output and error messages are sent to. And this is made possible by the “<” and “>” redirection operators.

How To Redirect Standard Output to File in Linux

You can redirect standard output as in the example below, here, we want to store the output of the top command for later inspection:

```
$ top -bn 5 >top.log
```

Where the flags:

- **-b** – enables **top** to run in batch mode, so that you can redirect its output to a file or another command.
- **-n** – specifies the number of iterations before the command terminates.

You can view the contents of **top.log** file using [cat command](#) as follows:

```
$ cat top.log
```

To append the output of a command, use the “>>” operator.

For instance to append the output of top command above in the **top.log** file especially within a script (or on the command line), enter the line below:

```
$ top -bn 5 >>top.log
```

Note: Using the file descriptor number, the output redirect command above is the same as:

```
$ top -bn 5 1>top.log
```

How To Redirect Standard Error to File in Linux

To redirect standard error of a command, you need to explicitly specify the file descriptor number, 2 for the shell to understand what you are trying to do.

For example the ls command below will produce an error when executed by a normal system user without root privileges:

```
$ ls -l /root/
```

You can redirect the standard error to a file as below:

```
$ ls -l /root/ 2>ls-error.log
```

```
$ cat ls-error.log
```

```
aaronkilik@tecmint ~ $ ls -l /root/
ls: cannot open directory '/root/': Permission denied
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $ ls -l /root/ 2>ls-error.log
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $ cat ls-error.log
ls: cannot open directory '/root/': Permission denied
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $
```

Redirect

Standard Error to File

In order to append the standard error, use the command below:

```
$ ls -l /root/ 2>>ls-error.log
```

How To Redirect Standard Output/ Error To One File

It is also possible to capture all the output of a command (both standard output and standard error) into a single file. This can be done in two possible ways by specifying the file descriptor numbers:

1. The first is a relatively old method which works as follows:

```
$ ls -l /root/ >ls-error.log 2>&1
```

The command above means the shell will first send the output of the `ls` command to the file **ls-error.log** (using `>ls-error.log`), and then writes all error messages to the file descriptor **2** (standard output) which has been redirected to the file **ls-error.log** (using `2>&1`). Implying that standard error is also sent to the same file as standard output.

2. The second and direct method is:


```
$ ls -l /root/ &>ls-error.log
```

You can as well append standard output and standard error to a single file like so:

```
$ ls -l /root/ &>>ls-error.log
```

How To Redirect Standard Input to File

Most if not all commands get their input from standard input, and by default standard input is attached to the keyboard.

To redirect standard input from a file other than the keyboard, use the “<” operator as below:

```
$ cat <domains.list
```

```
aaronkilik@tecmint ~ $ cat <domains.list
tecmint.com
news.tecmint.com
linuxsay.com
windowsmint.com
aaronkilik@tecmint ~ $
```

Redirect Standard Input to File

How To Redirect Standard Input/Output to File

You can perform standard input, standard output redirection at the same time using sort command as below:

```
$ sort <domains.list >sort.output
```

How to Use I/O Redirection Using Pipes

To redirect the output of one command as input of another, you can use pipes, this is a powerful means of building useful command lines for complex operations.

For example, the command below will list the top five recently modified files.

```
$ ls -lt | head -n 5
```

Here, the options:

- `-l` – enables long listing format
- `-t` – sort by modification time with the newest files are shown first
- `-n` – specifies the number of header lines to show

Important Commands for Building Pipelines

Here, we will briefly review two important commands for building command pipelines and they are:

xargs which is used to build and execute command lines from standard input. Below is an example of a pipeline which uses **xargs**, this command is used to copy a file into multiple directories in Linux:

```
$ echo /home/aaronkilik/test/ /home/aaronkilik/tmp | xargs -n 1 cp -v
/home/aaronkilik/bin/sys_info.sh
```

```
aaronkilik@tecmint ~ $ echo /home/aaronkilik/test/ /home/aaronkilik/tmp |
xargs -n 1 cp -v /home/aaronkilik/bin/sys_info.sh
'/home/aaronkilik/bin/sys_info.sh' -> '/home/aaronkilik/test/sys_info.sh'
'/home/aaronkilik/bin/sys_info.sh' -> '/home/aaronkilik/tmp/sys_info.sh'
aaronkilik@tecmint ~ $
```

Copy Files to Multiple Directories

And the options:

- `-n 1` – instructs xargs to use at most one argument per command line and send to the cp command
- `cp` – copies the file
- `-v` – displays progress of copy command.

For more usage options and info, read through the **xargs** man page:

```
$ man xargs
```

A **tee** command reads from standard input and writes to standard output and files. We can demonstrate how **tee** works as follows:

```
$ echo "Testing how tee command works" | tee file1
```

```
aaronkilik@tecmint ~ $ echo "Testing how tee command works" | tee file1
Testing how tee command works
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $ cat file1
Testing how tee command works
aaronkilik@tecmint ~ $
aaronkilik@tecmint ~ $
```

tee Command Example

File or text filters are commonly used with pipes for effective Linux file operations, to process information in powerful ways such as restructuring output of commands (this can be vital for generation of useful Linux reports), modifying text in files plus several other Linux system administration tasks.

Command Summary

- [cut](#) - remove sections from each line in a file
- [grep](#) - (search for and) print lines matching a pattern
- [sed](#) - command to modify a text stream
- [sort](#) - sort lines of text files
- [tr](#) - translate characters
- [uniq](#) - remove duplicate lines from a sorted file
- [wc](#) - print the number of bytes, words, and lines in files
- [pipe operator](#) - character (|) used to link the output of one command to the input of another
- [redirection](#) - changing the flow of an input, output or error stream

Variables in Linux

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 will 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"
```

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

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, the following script will access the value of defined variable NAME and print it on STDOUT –

[Live Demo](#)

```
#!/bin/sh

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

The above script will produce the following value –

```
Zara Ali
```

Read-only Variables

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

For example, the following script generates an error while trying to change the value of NAME –

[Live Demo](#)

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
```

```
NAME="Qadiri"
```

The above script will generate the following result –

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

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the unset command –

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

```
#!/bin/sh

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

The above example does 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 the command prompt.
- **Environment Variables** – An environment variable 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.

Shell Control Statements

Conditional Statements | Shell Script

Conditional Statements: There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows:

if statement

This block will process if specified condition is true.

Syntax:

```
if [ expression ]
then
    statement
fi
```

if-else statement

If specified condition is not true in if part then else part will be execute.

Syntax

```
if [ expression ]
then
    statement1
else
    statement2
fi
```

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

Syntax

```
if [ expression1 ]
then
    statement1
    statement2
    .
    .
elif [ expression2 ]
then
    statement3
    statement4
    .
    .
else
    statement5
fi
```

if..then..else..if..then..fi..fi..(Nested

if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

Syntax:

```

if [ expression1 ]
then
    statement1
    statement2
.
else
    if [ expression2 ]
    then
        statement3
    .
fi
fi

```

switch statement

case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern. When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed. If there is no match, the exit status of the case is zero.

Syntax:

```

case in
    Pattern 1) Statement 1;;
    Pattern n) Statement n;;
esac

```

Example Programs

Example 1:

Implementing if statement

```

#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
    echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
    echo "a is not equal to b"
fi

```

Output

```

$bash -f main.sh
a is not equal to b

```

Example 2:**Implementing if.else statement**

```
#Initializing two variables
a=20
b=20

if [ $a == $b ]
then
    #If they are equal then print this
    echo "a is equal to b"
else
    #else print this
    echo "a is not equal to b"
fi
```

Output

```
$bash -f main.sh
a is equal to b
```

Example 3:**Implementing switch statement**

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
    #case 1
    "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

    #case 2
    "audi") echo "Headquarters - Ingolstadt, Germany" ;;

    #case 3
    "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Output

```
$bash -f main.sh
Headquarters - Chennai, Tamil Nadu, India.
```

Looping Statements | Shell Script

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming

1. **while statement**
2. **for statement**
3. **until statement**

To alter the flow of loop statements, two commands are used they are,

1. **break**
2. **continue**

Their descriptions and syntax are as follows:

- **while statement**

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

Syntax

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

Example

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

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

Upon execution, you will receive the following result –

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

- **for statement**

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

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

-

Syntax

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

Syntax

- `for var in word1 word2 ... wordN`
- `do`
- `Statement(s) to be executed for every word.`
- `done`
- Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

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

```

• #!/bin/sh
•
• for var in 0 1 2 3 4 5 6 7 8 9
• do
•     echo $var
• done

```

- Upon execution, you will receive the following result –

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

- **until statement**

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax

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

Syntax

- until command
- do
- Statement(s) to be executed until command is true
- done
- Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

- **Example**

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

```

•  #!/bin/sh
•
•  a=0
•
•  until [ ! $a -lt 10 ]
•  do
•      echo $a
•      a=`expr $a + 1`
•  done

```

Upon execution, you will receive the following result –

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

Nesting Loops

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

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way –

Nesting while Loops

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

Syntax

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

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

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

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine –

```
#!/bin/sh

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

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

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

Example Programs

Example 1:

Implementing for loop with break statement

- php

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a is equal to 5 break the loop
    if [ $a == 5 ]
    then
        break
    fi
    # Print the value
    echo "Iteration no $a"
done
```

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
```

Example 2:

Implementing for loop with continue statement

- php

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a = 5 then continue the loop and
    # don't move to line 8
    if [ $a == 5 ]
    then
        continue
    fi
    echo "Iteration no $a"
done
```

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
Iteration no 6
Iteration no 7
Iteration no 8
Iteration no 9
Iteration no 10
```

Example 3:

Implementing while loop

- php

```
a=0
# -lt is less than operator

#Iterate the loop until a less than 10
while [ $a -lt 10 ]
do
    # Print the values
    echo $a

    # increment the value
    a=`expr $a + 1`
done
```

Output:

```
$bash -f main.sh
0
1
2
3
4
5
6
7
8
9
```

Example 4:

Implementing until loop

- php

```
a=0
# -gt is greater than operator

#Iterate the loop until a is greater than 10
until [ $a -gt 10 ]
do
    # Print the values
    echo $a

    # increment the value
    a=`expr $a + 1`
done
```

Output:

```
$bash -f main.sh
0
1
2
3
4
5
6
7
8
9
10
```

Note: Shell scripting is a case-sensitive language, which means proper syntax has to be followed while writing the scripts.

Example 5:

- PHP

```
COLORS="red green blue"
```

```
# the for loop continues until it reads all the values from the COLORS
```

```
for COLOR in $COLORS
do
    echo "COLOR: $COLOR"
done
```

Output:

```
$bash -f main.sh
COLOR: red
COLOR: green
COLOR: blue
```

Example 6:

We can even access the positional parameters using loops

We execute a shell script named sample.sh using three parameters

Script Execution: main.sh sample1 sample2 sample3

We can access above three parameters using \$@

- PHP

```
echo "Executing script"
```

```
# the script is executed using the below command
# main.sh sample1 sample2 sample
# where sample1, sample2 and sample3 are the positional arguments
# here $@ contains all the positional arguments.
```

```
for SAMPLE in $@
do
    echo "The given sample is: $SAMPLE"
done
```

Output:

```
$bash -f main.sh sample1 sample2 sample3
Executing script
The given sample is sample1
The given sample is sample2
The given sample is sample3
```

Example 7: Infinite loop

- PHP

```
while true
do
    # Command to be executed
```



```
# sleep 1 indicates it sleeps for 1 sec
echo "Hi, I am infinity loop"
sleep 1
done
```

Output:

```
$bash -f main.sh
Hi, I am infinity loop
Hi, I am infinity loop
Hi, I am infinity loop
```

```
.
.
.
.
```

It continues

Example 8: Checking for user input

- PHP

```
CORRECT=n
while [ "$CORRECT" == "n" ]
do
    # loop discontinues when you enter y i.e.e, when your name is correct
    # -p stands for prompt asking for the input
    read -p "Enter your name:" NAME
    read -p "Is ${NAME} correct? " CORRECT
done
```

Output:

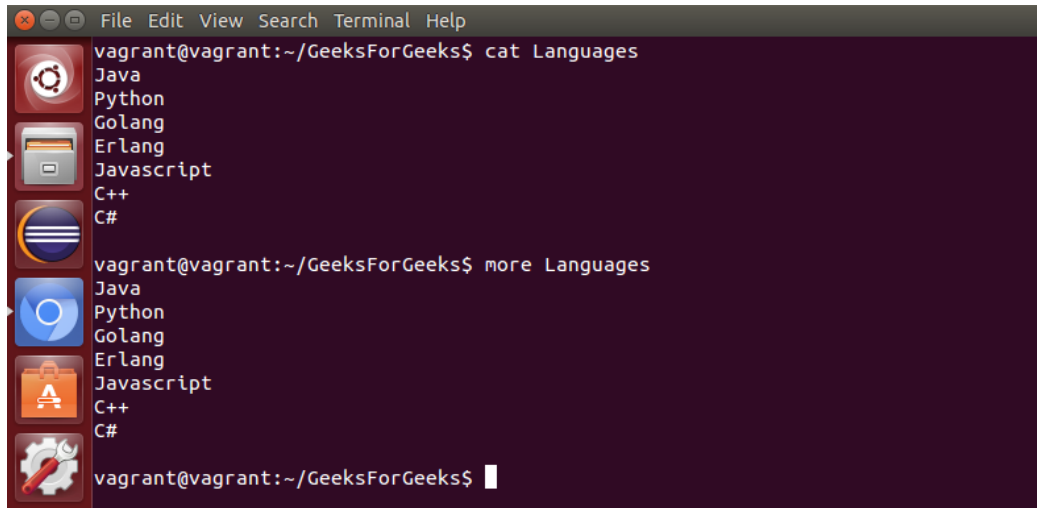
```
$bash -f main.sh
Enter your name:Ironman
Is Ironman correct? n
Enter your name:Spiderman
Is Spiderman correct? y
```

Basic Shell Commands in Linux

A [shell](#) is a special user program that provides an interface to the user to use operating system services. Shell accepts human-readable commands from the user and converts them into something which the kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.

1). Displaying the file contents on the terminal:

- [cat](#): It is generally used to concatenate the files. It gives the output on the standard output.
- [more](#): It is a filter for paging through text one screenful at a time.



```

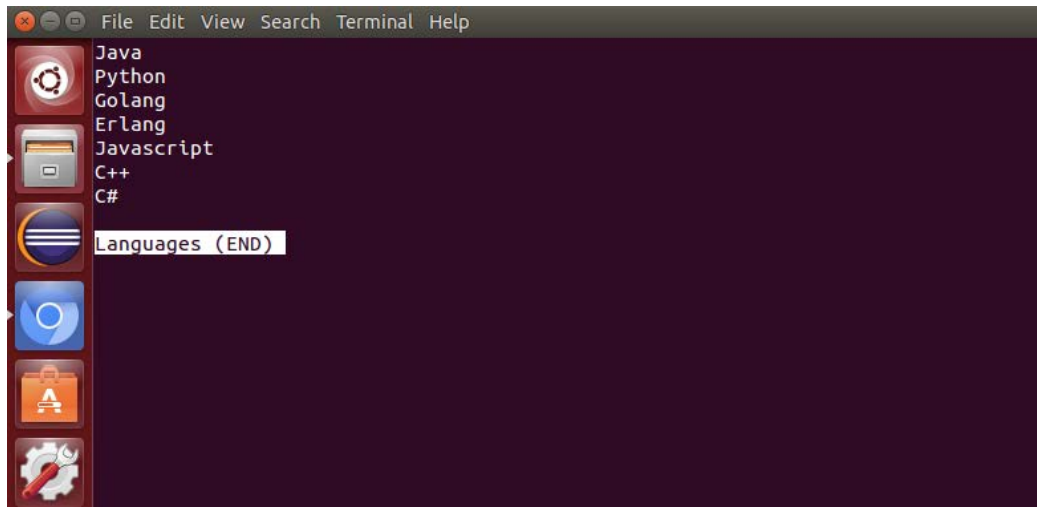
vagrant@vagrant:~/GeeksForGeeks$ cat Languages
Java
Python
Golang
Erlang
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$ more Languages
Java
Python
Golang
Erlang
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$

```

- [less](#): It is used to viewing the files instead of opening the file. Similar to *more* command but it allows backward as well as forward movement.

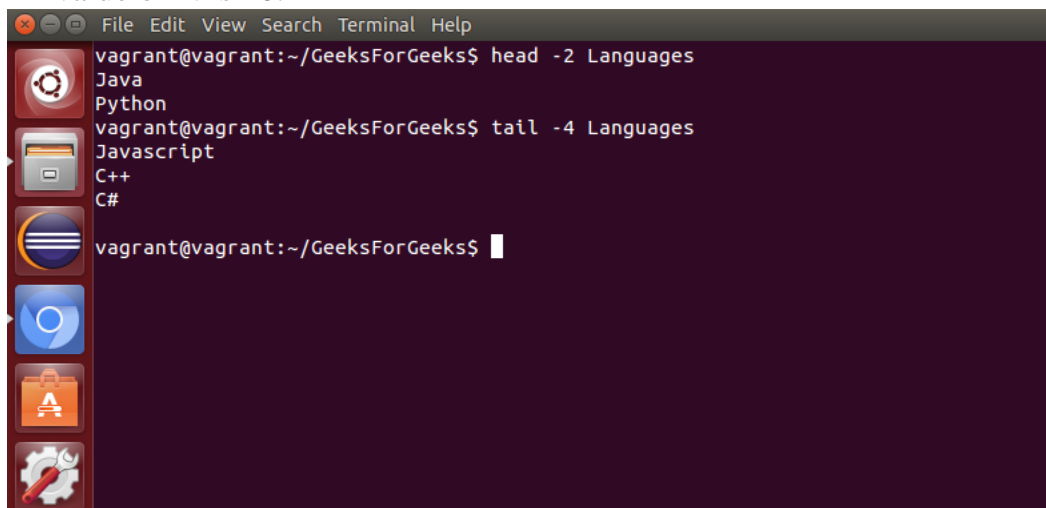


```

Java
Python
Golang
Erlang
Javascript
C++
C#
Languages (END)

```

- [head](#) : Used to print the first N lines of a file. It accepts N as input and the default value of N is 10.
- [tail](#) : Used to print the last N-1 lines of a file. It accepts N as input and the default value of N is 10.



```

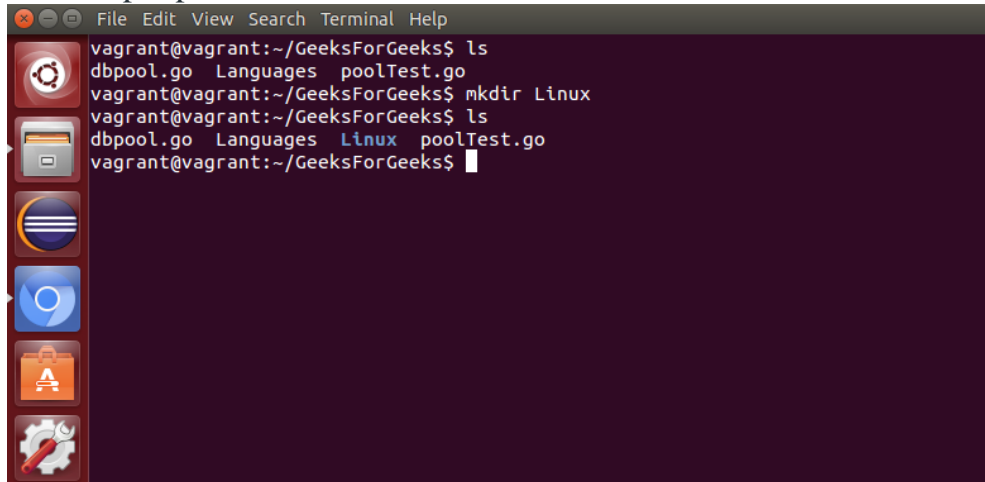
vagrant@vagrant:~/GeeksForGeeks$ head -2 Languages
Java
Python
vagrant@vagrant:~/GeeksForGeeks$ tail -4 Languages
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$

```

2). File and Directory Manipulation Commands:

- **mkdir** : Used to create a directory if not already exist. It accepts the directory name as an input parameter.

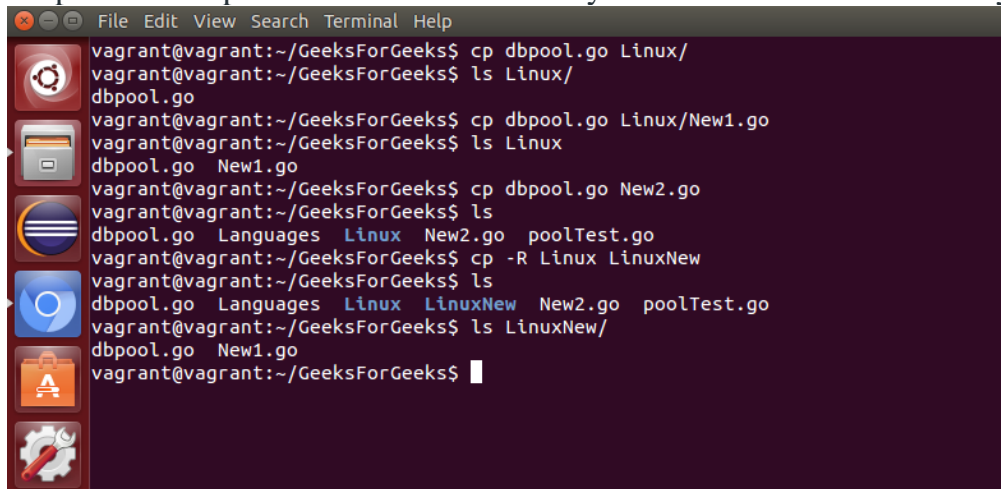


```

vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ mkdir Linux
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$

```

- **cp** : This command will copy the files and directories from the source path to the destination path. It can copy a file/directory with the new name to the destination path. It accepts the source file/directory and destination file/directory.

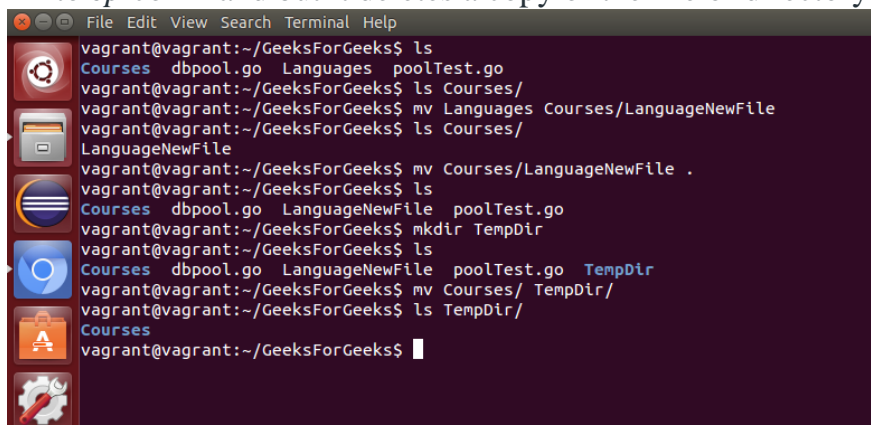


```

vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go Linux/
vagrant@vagrant:~/GeeksForGeeks$ ls Linux/
dbpool.go
vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go Linux/New1.go
vagrant@vagrant:~/GeeksForGeeks$ ls Linux
dbpool.go  New1.go
vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go New2.go
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ cp -R Linux LinuxNew
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  LinuxNew  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ ls LinuxNew/
dbpool.go  New1.go
vagrant@vagrant:~/GeeksForGeeks$

```

- **mv** : Used to move the files or directories. This command's working is almost similar to *cp* command but it deletes a copy of the file or directory from the source path.

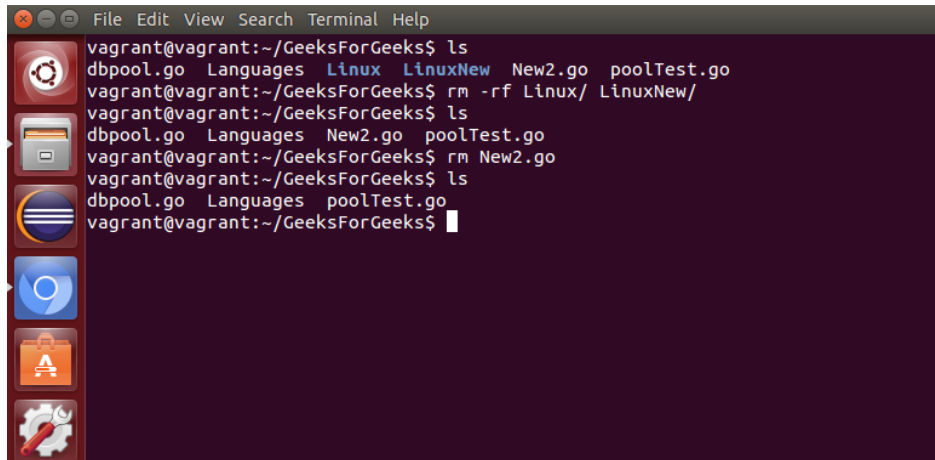


```

vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ ls Courses/
vagrant@vagrant:~/GeeksForGeeks$ mv Languages Courses/LanguageNewFile
vagrant@vagrant:~/GeeksForGeeks$ ls Courses/
LanguageNewFile
vagrant@vagrant:~/GeeksForGeeks$ mv Courses/LanguageNewFile .
vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  LanguageNewFile  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ mkdir TempDir
vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  LanguageNewFile  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$ mv Courses/ TempDir/
vagrant@vagrant:~/GeeksForGeeks$ ls TempDir/
Courses
vagrant@vagrant:~/GeeksForGeeks$

```

- **rm** : Used to remove files or directories.

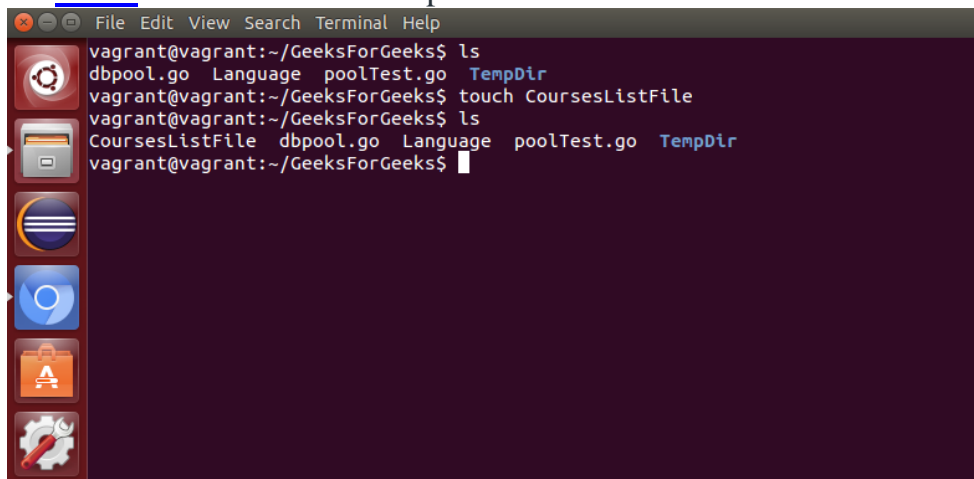


```

vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  LinuxNew  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ rm -rf Linux/ LinuxNew/
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ rm New2.go
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$

```

- **touch** : Used to create or update a file.



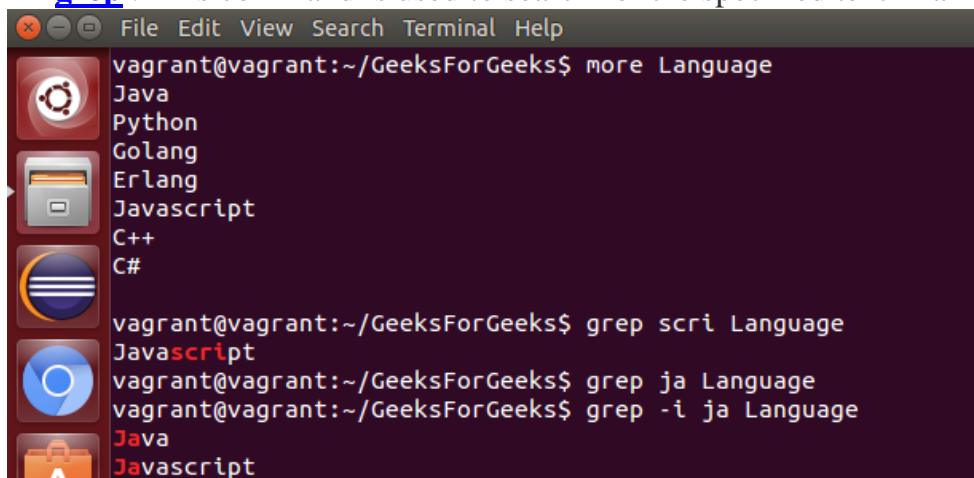
```

vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Language  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$ touch CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ ls
CoursesListFile  dbpool.go  Language  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$

```

3). Extract, sort, and filter data Commands:

- **grep** : This command is used to search for the specified text in a file.

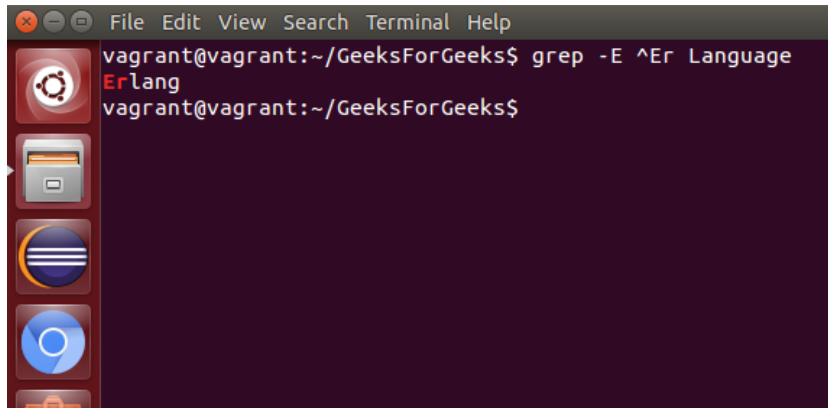


```

vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
Golang
Erlang
Javascript
C++
C#
vagrant@vagrant:~/GeeksForGeeks$ grep scri Language
Javascrip
vagrant@vagrant:~/GeeksForGeeks$ grep ja Language
vagrant@vagrant:~/GeeksForGeeks$ grep -i ja Language
Java
Javascript

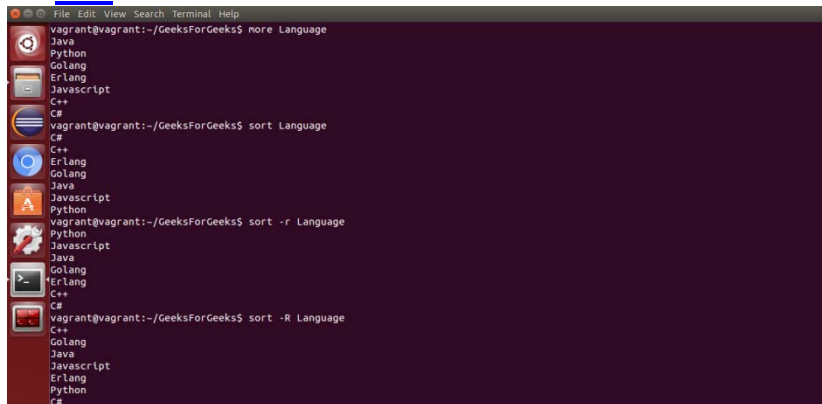
```

- **grep with Regular Expressions**: Used to search for text using specific regular expressions in file.



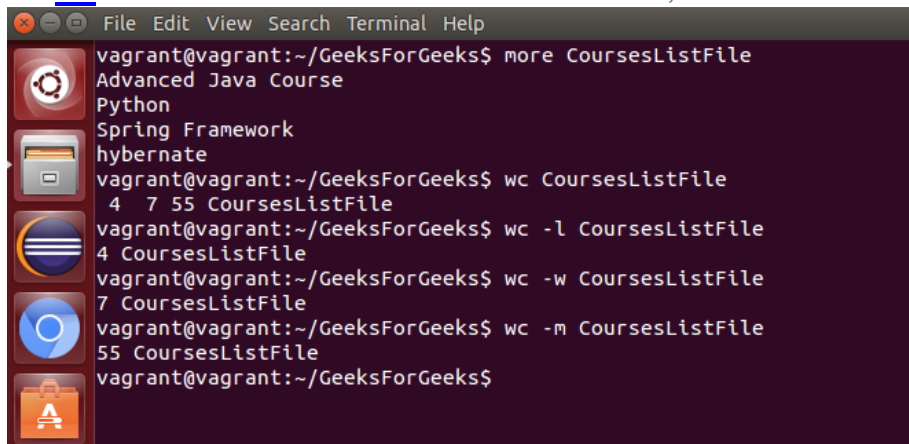
```
vagrant@vagrant:~/GeeksForGeeks$ grep -E ^Er Language
Er
lang
vagrant@vagrant:~/GeeksForGeeks$
```

- **sort** : This command is used to sort the contents of files.



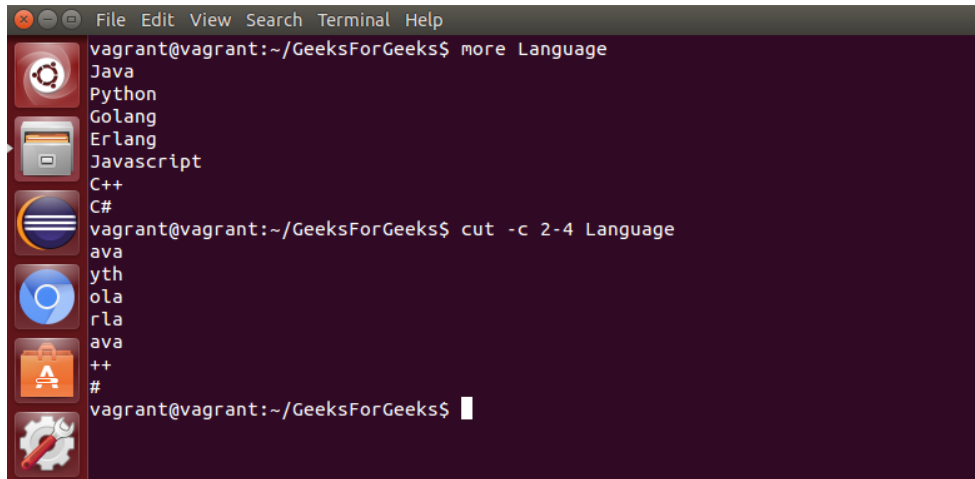
```
vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
GoLang
Erlang
JavaScript
C#
C++
vagrant@vagrant:~/GeeksForGeeks$ sort Language
C#
C++
C
Erlang
GoLang
Java
JavaScript
Python
vagrant@vagrant:~/GeeksForGeeks$ sort -r Language
Python
JavaScript
Java
GoLang
Erlang
C#
C++
vagrant@vagrant:~/GeeksForGeeks$ sort -R Language
C++
GoLang
Java
JavaScript
Erlang
Python
C#
```

- **wc** : Used to count the number of characters, words in a file.



```
vagrant@vagrant:~/GeeksForGeeks$ more CoursesListFile
Advanced Java Course
Python
Spring Framework
hibernate
vagrant@vagrant:~/GeeksForGeeks$ wc CoursesListFile
4 7 55 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -l CoursesListFile
4 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -w CoursesListFile
7 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -m CoursesListFile
55 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$
```

- **cut** : Used to cut a specified part of a file.



```

vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
Golang
Erlang
Javascript
C++
C#
vagrant@vagrant:~/GeeksForGeeks$ cut -c 2-4 Language
ava
yth
ola
rla
ava
++
#
vagrant@vagrant:~/GeeksForGeeks$

```

4). Basic Terminal Navigation Commands:

- [ls](#) : To get the list of all the files or folders.
- **ls -l**: Optional flags are added to **ls** to modify default behavior, listing contents in extended form **-l** is used for “**long**” output
- **ls -a**: Lists of all files including the hidden files, add **-a** flag
- [cd](#): Used to change the directory.
- [du](#): Show disk usage.
- [pwd](#): Show the present working directory.
- [man](#): Used to show the manual of any command present in Linux.
- [rmdir](#): It is used to delete a directory if it is empty.
- [ln file1 file2](#): Creates a physical link.
- [ln -s file1 file2](#): Creates a symbolic link.
- [locate](#): It is used to locate a file in Linux System
- [echo](#): This command helps us move some data, usually text into a file.
- [df](#): It is used to see the available disk space in each of the partitions in your system.
- [tar](#): Used to work with tarballs (or files compressed in a tarball archive)

5). **File Permissions Commands**: The *chmod* and *chown* commands are used to control access to files in UNIX and Linux systems.

- [chown](#) : Used to change the owner of the file.
- [chgrp](#) : Used to change the group owner of the file.
- [chmod](#) : Used to modify the access/permission of a user.

Compiling Testing & Debugging

Testing is an integral and important part of any software development cycle, open or closed, and Linux kernel is no exception to that. Developer testing, integration testing, regression, and stress testing have different individual goals, however from 1000 feet up, the end goal is the same, to ensure the software continues to work as it did before adding a new body of code, and the new features work as designed.

Basic Testing

Once a new kernel is installed, the next step is try to boot it and see what happens. Once the new kernel is up and running, check dmesg for any regressions. Run a few usage tests:

- Is networking (wifi or wired) functional?
- Does ssh work?
- Run rsync of a large file over ssh
- Run git clone and git pull
- Start web browser
- Read email
- Download files: ftp, wget etc.
- Play audio/video files
- Connect new USB devices - mouse, usb stick etc.

Examine Kernel Logs

Checking for regressions in dmesg is a good way to identify problems, if any, introduced by the new code. As a general rule, there should be no new crit, alert, and emerg level messages in dmesg. There should be no new err level messages. Pay close attention to any new warn level messages as well. Please note that new warn messages aren't as bad. New code at times adds new warning messages which are just warnings.

- `dmesg -t -l emerg`
- `dmesg -t -l crit`
- `dmesg -t -l alert`
- `dmesg -t -l err`
- `dmesg -t -l warn`
- `dmesg -t -k`
- `dmesg -t`

The following script runs the above dmesg commands and saves the output for comparing with older release dmesg files. It then runs diff commands against the older release dmesg files. Old release is a required input parameter. If one is not supplied, it will simply generate dmesg files and exit. Regressions indicate newly introduced bugs and/or bugs that escaped patch testing and integration testing in linux git trees prior to including the patch in a release. Are there any stack traces resulting from WARN_ON in the dmesg? These are serious problems that require further investigation.

- [dmesg regression check script](#)

Stress Testing

Running 3 to 4 kernel compiles in parallel is a good overall stress test. Download a few Linux kernel gits, stable, linux-next etc.. Run timed compiles in parallel. Compare times with old runs of this test for regressions in performance. Longer compile times could be indicators of performance regression in one of the kernel modules. Performance problems are hard to debug. First step is to detect them. Running several compiles in parallel is a good overall stress test that could be used as a performance regression test and overall kernel regression test, as it exercises various kernel modules like memory, file-systems, dma, and drivers.

```
time make all
```

Kernel Testing Tools

There are several tests under tools/testing that are included in the Linux kernel git. There is a good mix of automated and functional tests.

ktest suite

ktest is an automated test suite that can test builds, installs, and kernel boots. It can also run cross-compile tests provided the system has cross-compilers installed. ktest depends on flex and bison tools. Please consult the ktest documentation in tools/testing/ktest for details on how to run ktest. It is left to the reader as a self-study. A few resources that go into detail on how to run ktest:

- ktest-eLinux.org

tools/testing/selftests

Let's start with selftests. Kernel sources include a set of self-tests which test various sub-systems. As of this writing, breakpoints, cpu-hotplug, efivarfs, ipc, kcmp, memory-hotplug, mqueue, net, powerpc, ptrace, rcutorture, timers, and vm sub-systems have self-tests. In addition to these, user memory self-tests test user memory to kernel memory copies via test_user_copy module. The following is on how to run these self-tests:

Compile tests:

```
make -C tools/testing/selftests
```

Run all tests: (running some tests needs root access, login as root and run)

```
make -C tools/testing/selftests run_tests
```

Run only tests targeted for a single sub-system:

```
make -C tools/testing/selftests TARGETS=vm run_tests
```

tools/testing/fault-injection

Another test suite under tools/testing is fault-injection. failcmd.sh script runs a command to inject slab and page allocation failures. This type of testing helps validate how well kernel can recover from faults. This test should be run as root. The following is a quick summary of currently implemented fault injection capabilities. The list keeps growing as new fault injection capabilities get added. Please refer to the Documentation/fault-injection/fault-injection.txt for the latest.

failslab (default option)

injects slab allocation failures. kmalloc(), kmem_cache_alloc(), ...

fail_page_alloc

injects page allocation failures. `alloc_pages()`, `get_free_pages()`, ...

fail_make_request

injects disk IO errors on devices permitted by setting, `/sys/block//make-it-fail` or `/sys/block///make-it-fail`. (`generic_make_request()`)

fail_mmc_request

injects MMC data errors on devices permitted by setting debugfs entries under `/sys/kernel/debug/mmc0/fail_mmc_request`

The capabilities and behavior of fault-injection can be configured. `fault-inject-debugfs` kernel module provides some debugfs entries for runtime. Ability to specify the error probability rate for faults, the interval between fault injection are just a couple of examples of the configuration choices `fault-injection` test supports. Please refer to the `Documentation/fault-injection/fault-injection.txt` for details. Boot options can be used to inject faults during early boot before debugfs becomes available. The following boot options are supported:

- `failslab=`
- `fail_page_alloc=`
- `fail_make_request=`
- `mmc_core.fail_request=[interval],[probability],[space],[times]`

The fault-injection infrastructure provides interfaces to add new fault-injection capabilities. The following is a brief outline of the steps involved in adding a new capability. Please refer to the above mentioned document for details:

define the fault attributes using `DECLARE_FAULT_INJECTION(name);`

Please see the definition of struct `fault_attr` in `fault-inject.h` for details.

add a boot option to configure fault attributes

This can be done using helper function `setup_fault_attr(attr, str);` Adding a boot option is necessary to enable the fault injection capability during early boot time.

add debugfs entries

Use the helper function `fault_create_debugfs_attr(name, parent, attr);` to add new debugfs entries for this new capability.

add module parameters

Adding module parameters to configure the fault attributes is a good option, when the scope of the new fault capability is limited to a single kernel module.

add a hook to insert failures

`should_fail(attr, size);` Upon `should_fail()` returning true, client code should inject a failure.

Applications using this fault-injection infrastructure can target a specific kernel module to inject slab and page allocation failures to limit the testing scope if need be.

Auto Testing Tools

There are several automated testing tools and test infrastructures that you can chose from based on your specific testing needs. This section is intended to be a brief overview and not a detailed guide on how to use each of these.

AuToTest

Autotest is a framework for fully automated testing. It is designed primarily to test the Linux kernel, though it is useful for many other functions such as qualifying new hardware. It is an open source project under the GPL. Autotest works in server-client mode. Autotest server can be configured to initiate, run, and monitor tests on several target systems running the autotest client. Autotest client can be run manually on a target system or via the server. Using this framework, new test cases can be added. Please Autotest White Paper for more information.

Linaro Automated Validation Architecture

LAVA-Test Automated Testing Framework is a framework to help with automated installation and executions of tests. For example, running LTP in LAVA framework can be accomplished with a few commands. Running lava-test tool to install LTP will automatically install any dependencies, download the source for the recent release of LTP, compile it, and install the binaries in a self-contained area so that they can be removed easily when user runs uninstall. At this point running lava-test run with ltp test option will execute LTP tests and save results with an unique id that includes the test name, time/date stamp of the test execution. These results are saved for future reference. This is a good feature to find regressions, if any, between test runs. Summary of commands to run as an example:

Show a list of tests supported by lava-test:

```
lava-test list-tests
```

Install a new test:

```
lava-test install ltp
```

Run the test:

```
lava-test run ltp
```

Check results:

```
lava-test results show ltp-timestamp.0
```

Remove tests:

```
lava-test uninstall ltp
```

Kernel Debug Features

Linux kernel includes several debugging features such as kmemcheck and kmemleak.

kmemcheck

kmemcheck is a dynamic checking tool that detects and warns about some uses of uninitialized memory. It serves the same function as Valgrind's memcheck which is a userspace memory checker, where as kmemcheck checks kernel memory. CONFIG_KMEMCHECK kernel configuration option enables the kmemcheck debugging feature. Please read the Documentation/kmemcheck.txt for information on how to configure and use this feature, and how to interpret the reported results.

kmemleak

kmemleak can be used to detect possible kernel memory leaks in a way similar to a tracing garbage collector. The difference between the tracing garbage collector and kmemleak is that the latter doesn't free orphan objects, instead it reports them in /sys/kernel/debug/kmemleak. A similar method of reporting and not freeing is used by the Valgrind's memcheck --leak-check to detect memory leaks in user-space applications. CONFIG_DEBUG_KMEMLEAK kernel configuration option enables the kmemleak debugging feature. Please read the Documentation/kmemleak.txt for information on how to configure and use this feature, and how to interpret the reported results.

Kernel Debug Interfaces

Linux kernel has support for static and dynamic debugging via configuration options, debug APIs, interfaces, and frameworks. Let's learn more about each of these starting with the static options.

Debug Configuration Options - Static

Linux kernel core and several Linux kernel modules, if not all, include kernel configuration options to debug. Several of these static debug options can be enabled at compile time. Debug messages are logged in dmesg buffer.

Debug APIs

An example of Debug APIs is DMA-debug which is designed for debugging driver dma api usage errors. When enabled, it keeps track of dma mappings per device, detects unmap attempts on addresses that aren't mapped, and missing mapping error checks in driver code after dma map attempts. CONFIG_HAVE_DMA_API_DEBUG and CONFIG_DMA_API_DEBUG kernel configuration options enable this feature on architectures that provide the support. With the CONFIG_DMA_API_DEBUG option enabled, the Debug-dma interfaces are called from DMA

API. For example, when a driver calls `dma_map_page()` to map a dma buffer, `dma_map_page()` will call `debug_dma_map_page()` to start tracking the buffer until it gets released via `dma_unmap_page()` at a later time. For further reading on Detecting silent data corruptions and memory leaks using DMA Debug API

Dynamic Debug

Dynamic debug feature allows dynamically enabling/disabling `pr_debug()`, `dev_dbg()`, `print_hex_dump_debug()`, `print_hex_dump_bytes()` per-callsite. What this means is, a specific debug message can be enabled at run-time to learn more about a problem that is observed. This is great because, there is no need to re-compile the kernel with debug options enabled, then install the new kernel, only to find that the problem is no longer reproducible. Once `CONFIG_DYNAMIC_DEBUG` is enabled in the kernel, dynamic debug feature enables a fine grain enable/disable of debug messages. `/sys/kernel/debug/dynamic_debug/control` is used to specify which `pr_*` messages are enabled. A quick summary of how to enable dynamic debug per call level, per module level is as follows:

Enable `pr_debug()` in `kernel/power/suspend.c` at line 340:

```
echo 'file suspend.c line 340 +p' > /sys/kernel/debug/dynamic_debug/control
```

Enable dynamic debug feature in a module at module load time

Pass in `dyndbg="plmft"` to `modprobe` at the time module is being loaded.

Enable dynamic debug feature in a module to persist across reboots

create or change `modname.conf` file in `/etc/modprobe.d/` to add `dyndbg="plmft"` option. However for drivers that get loaded from `initramfs`, changing `modname.conf` is insufficient for the dynamic debug feature to persist across reboot. For such drivers, change `grub` to pass in `module.dyndbg="+plmft"` as a module option as a kernel boot parameter.

`dynamic_debug.verbose=1` kernel boot option increases the verbosity of dynamic debug messages. Please consult the `Documentation/dynamic-debug-howto.txt` for more information on this feature.

UNIT 5

File Ownership, Access and Permission

File Permissions

All the three owners (user owner, group, others) in the Linux system have three types of permissions defined. Nine characters denotes the three types of permissions.

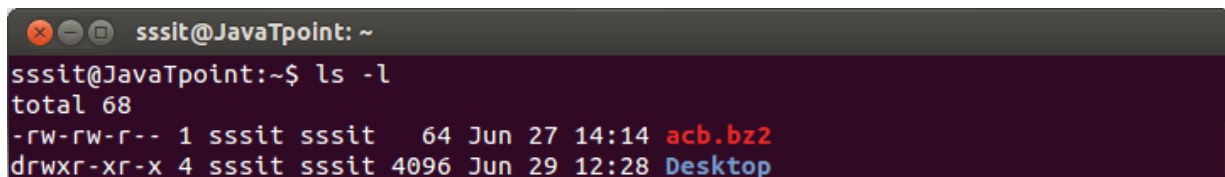
1. **Read (r)** : The read permission allows you to open and read the content of a file. But you can't do any editing or modification in the file.

2. **Write (w)** : The write permission allows you to edit, remove or rename a file. For instance, if a file is present in a directory, and write permission is set on the file but not on the directory, then you can edit the content of the file but can't remove, or rename it.
3. **Execute (x)**: In Unix type system, you can't run or execute a program unless execute permission is set. But in Windows, there is no such permission available.

Permissions are listed below:

permission	on a file	on a directory
r (read)	read file content (cat)	read directory content (ls)
w (write)	change file content (vi)	create file in directory (touch)
x (execute)	execute the file	enter the directory (cd)

Permission Set



```

sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ ls -l
total 68
-rw-rw-r-- 1 sssit sssit  64 Jun 27 14:14 acb.bz2
drwxr-xr-x 4 sssit sssit 4096 Jun 29 12:28 Desktop

```

Look at the above snapshot, there are ten characters (-rw-rw-r--) before the user owner. We'll describe these ten characters here.

File permissions for (-rw-rw-r--)

position	characters	ownership
1	-	denotes file type
2-4	rw-	permission for user
5-7	rw-	permission for group
8-10	r--	permission for other

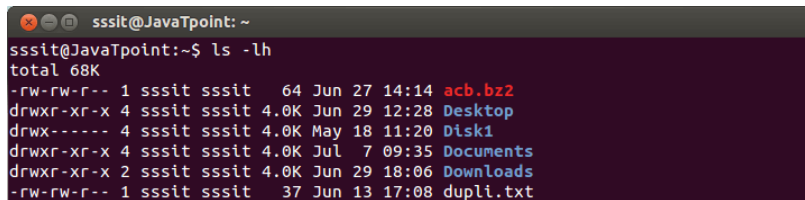
When you are the **User owner**, then the user owner permission applies to you. Other permissions are not relevant to you.

When you are the **Group** then the group permission applies to you. Other permissions are not relevant to you.

When you are the **Other**, then the other permission applies to you. User and group permissions are not relevant to you.

Permission Example

Now we'll show some examples how permissions can be seen for a file or directory.



```

ssst@JavaTpoint: ~
ssst@JavaTpoint:~$ ls -lh
total 68K
-rw-rw-r-- 1 ssst ssst 64 Jun 27 14:14 acb.bz2
drwxr-xr-x 4 ssst ssst 4.0K Jun 29 12:28 Desktop
drwx----- 4 ssst ssst 4.0K May 18 11:20 Disk1
drwxr-xr-x 4 ssst ssst 4.0K Jul 7 09:35 Documents
drwxr-xr-x 2 ssst ssst 4.0K Jun 29 18:06 Downloads
-rw-rw-r-- 1 ssst ssst 37 Jun 13 17:08 dupli.txt

```

Look at the above snapshot, different directories and files have different permissions.

First letter (-) or **d** represents the files and directories respectively.

Now, from remaining nine letters, **first** triplet represents the permission for **user owner**. Second triplet represents the permission for **group owner**. **Third** triplet represents the permission for **other**.

Linux File System Overview

Let's take a moment and have an overview of the main file types:

1. Regular Files

These are the most common file types. Regular files contain human-readable text, program instructions, and ASCII characters.

Examples of regular files include:

- Simple text files, pdf files
- Multimedia files such as image, music, and video files
- Binary files
- Zipped or compressed files

And so much more.

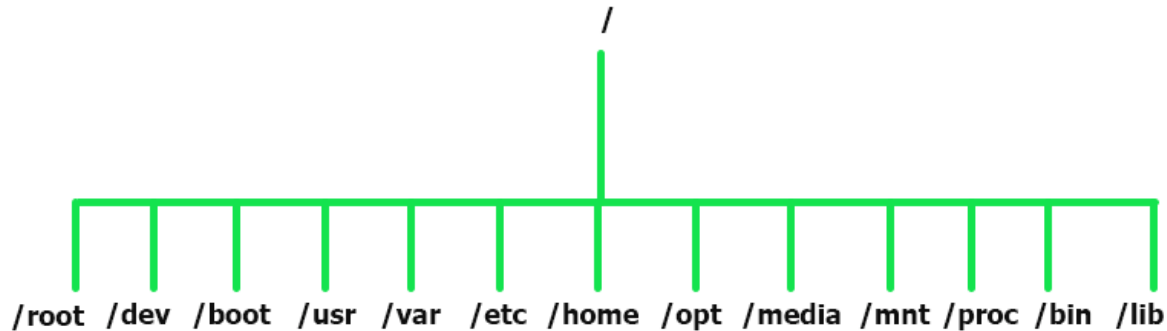
2. Special Files

These are files that represent physical devices such as mounted volumes, printers, CD drives, and any I/O) input and output device.

3. Directories

A **directory** is a special file type that stores both regular and special files in a hierarchical order starting from the root (/) directory. A directory is the equivalent of a folder in the Windows operating system. Directories are created using the **mkdir** command, short for making the directory, as we shall see later on in this tutorial.

The Linux hierarchy structure starts from the root directory and branches out to other directories as shown:



Linux Directory Structure

Let's understand each directory and its usage.

- The **/root** directory is the home directory for the root user.
- The **/dev** directory contains device files such as **/dev/sda**.
- Static boot files are located in the **/boot** directory.
- Applications and user utilities are found in the **/usr** directory.
- The **/var** directory contains log files of various system applications.
- All system configuration files are stored in the **/etc** directory.
- The **/home** directory is where user folders are located. These include Desktop, Documents, Downloads, Music, Public, and Videos.
- For add-on application packages, check them out in the **/opt** directory.
- The **/media** directory stores files for removable devices such as USB drives.
- The **/mnt** directory contains subdirectories that act as temporary mount points for mounting devices such as CD-ROMs.
- The **/proc** directory is a virtual filesystem that holds information on currently running processes. It's a strange filesystem that is created upon a system boot and destroyed upon shutdown.
- The **/bin** directory contains user command binary files.
- The **/lib** directory stores shared library images and kernel modules.

Linux File Management Commands

You will spend a great deal of time interacting with the terminal where you will be running commands. Executing commands is the most preferred way of interacting with a Linux system as it gives you total control over the system compared to using the graphical display elements.

For this lesson, and the coming lessons, we will be running commands on the terminal. We are using **Ubuntu OS** and to launch the terminal, use the keyboard shortcut CTRL + ALT + T.

Let's now delve into the basic file management commands that will help you create and manage your files on your system.

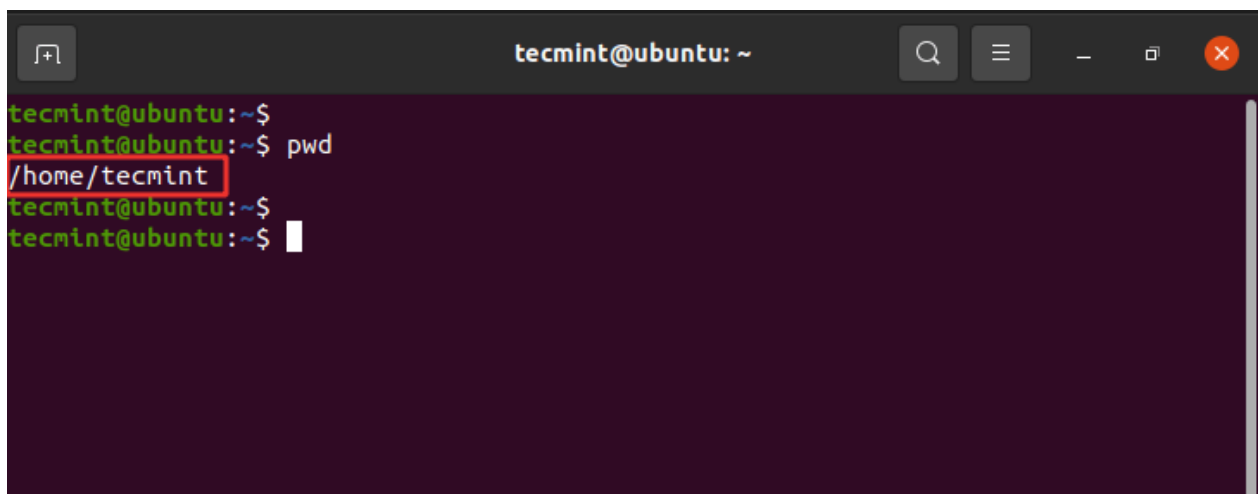
1. pwd Command

pwd, short for the print working directory, is a command that prints out the current working directory in a hierarchical order, beginning with the topmost root directory (/).

To check your current working directory, simply invoke the **pwd** command as shown.

```
$ pwd
```

The output shows that we are in our home directory, the absolute or full path being **/home/tecmint**.

A terminal window titled 'tecmint@ubuntu: ~' with search, menu, and window control icons. The terminal shows the following sequence: 'tecmint@ubuntu:~\$' followed by 'tecmint@ubuntu:~\$ pwd' where 'pwd' is highlighted in green. The output is '/home/tecmint', which is highlighted with a red box. The prompt then returns to 'tecmint@ubuntu:~\$' and then to 'tecmint@ubuntu:~\$' with a cursor.

Print Current Working Directory

2. cd Command

To change or navigate directories, use the **cd** command which is short for change directory.

For instance, to navigate to the **/var/log** file path, run the command:

```
$ cd /var/log
```



```

tecmint@ubuntu: /var/log
tecmint@ubuntu:/$
tecmint@ubuntu:/$ cd /var/log
tecmint@ubuntu:/var/log$
tecmint@ubuntu:/var/log$ pwd
/var/log
tecmint@ubuntu:/var/log$
tecmint@ubuntu:/var/log$

```

Navigate Directories in Linux

To go a directory up append two dots or periods in the end.

```
$ cd ..
```

To go back to the home directory run the cd command without any arguments.

```
$ cd
```

```

tecmint@ubuntu: ~
tecmint@ubuntu:/$
tecmint@ubuntu:/$ cd /var/log
tecmint@ubuntu:/var/log$
tecmint@ubuntu:/var/log$ pwd
/var/log
tecmint@ubuntu:/var/log$
tecmint@ubuntu:/var/log$ cd ..
tecmint@ubuntu:/var$
tecmint@ubuntu:/var$
tecmint@ubuntu:/var$ cd
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$ pwd
/home/tecmint
tecmint@ubuntu:~$

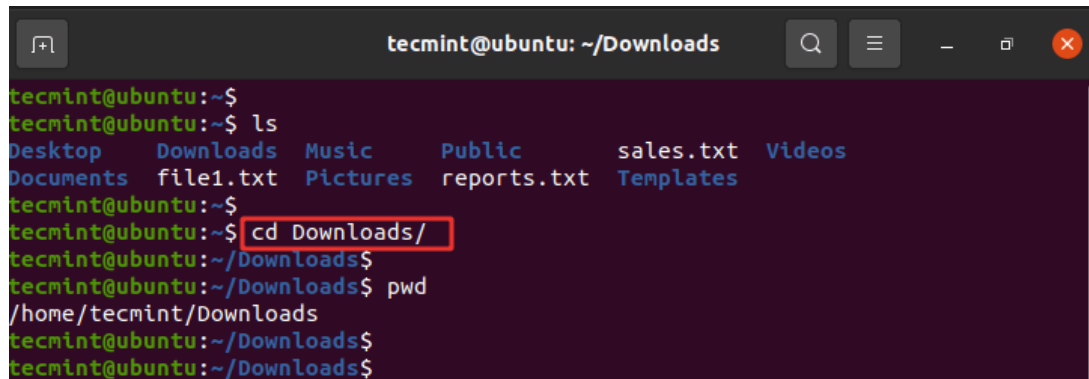
```

cd Command Examples

NOTE: To navigate into a subdirectory or a directory within your current directory, don't use a forward slash (/) simply type in the name of the directory.

For example, to navigate into the Downloads directory, run:

```
$ cd Downloads
```



```

tecmint@ubuntu: ~/Downloads
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  Videos
Documents file1.txt  Pictures   reports.txt Templates
tecmint@ubuntu:~$ cd Downloads/
tecmint@ubuntu:~/Downloads$ pwd
/home/tecmint/Downloads
tecmint@ubuntu:~/Downloads$
tecmint@ubuntu:~/Downloads$

```

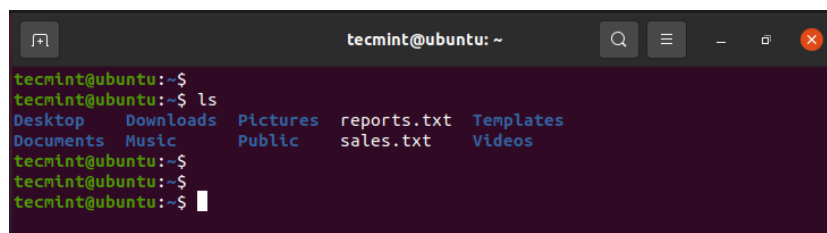
Navigate to Downloads Directory

3. ls Command

The `ls` command is a command used for listing existing files or folders in a directory. For example, to list all the contents in the home directory, we will run the command.

```
$ ls
```

From the output, we can see that we have two text files and eight folders which are usually created by default after installing and logging in to the system.



```

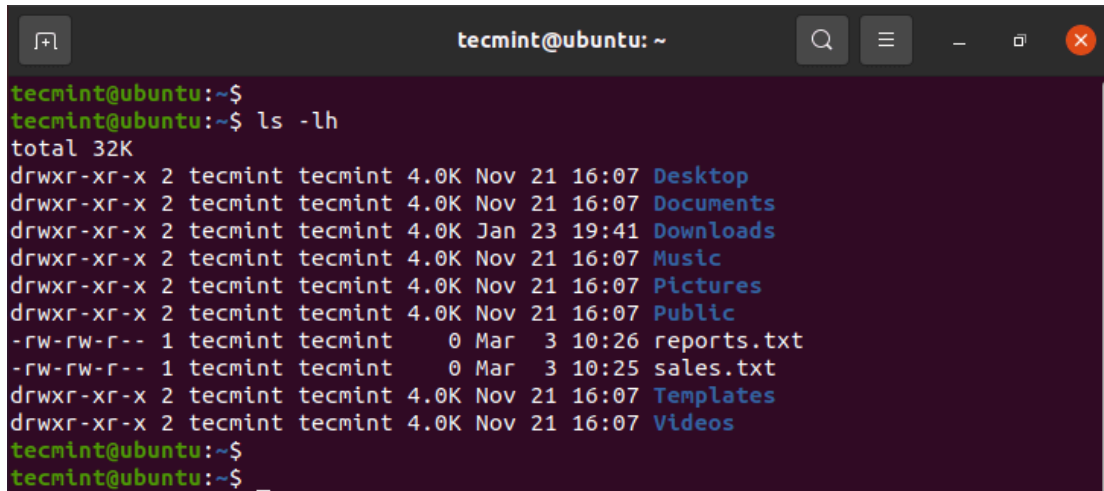
tecmint@ubuntu: ~
tecmint@ubuntu:~$ ls
Desktop  Downloads  Pictures  reports.txt  Templates
Documents Music      Public    sales.txt    Videos
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

List Files in Linux

To list more information append the `-lh` flag as shown. The `-l` option stands for long listing and prints out additional information such as file permissions, user, group, file size, and date of creation. The `-h` flag prints out the file or directory size in a human-readable format.

```
$ ls -lh
```



```
tecmint@ubuntu: ~  
tecmint@ubuntu:~$  
tecmint@ubuntu:~$ ls -lh  
total 32K  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Desktop  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Documents  
drwxr-xr-x 2 tecmint tecmint 4.0K Jan 23 19:41 Downloads  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Music  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Pictures  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Public  
-rw-rw-r-- 1 tecmint tecmint  0 Mar  3 10:26 reports.txt  
-rw-rw-r-- 1 tecmint tecmint  0 Mar  3 10:25 sales.txt  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Templates  
drwxr-xr-x 2 tecmint tecmint 4.0K Nov 21 16:07 Videos  
tecmint@ubuntu:~$  
tecmint@ubuntu:~$
```

Long List Files in Linux

To list hidden files, append the `-a` flag.

```
$ ls -la
```

This displays hidden files which start with a period sign `(.)` as shown.

```
.ssh  
  
.config  
  
.local
```

```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ ls -la
total 92
drwxr-xr-x 18 tecmint tecmint 4096 Mar  3 10:26 .
drwxr-xr-x  9 root    root    4096 Mar  2 15:36 ..
-rw-r--r--  1 tecmint tecmint 2003 Jan 23 20:15 .bash_history
-rw-r--r--  1 tecmint tecmint 220  Aug 29 2020 .bash_logout
-rw-r--r--  1 tecmint tecmint 3771 Aug 29 2020 .bashrc
drwx----- 14 tecmint tecmint 4096 Jan 23 19:31 .cache
drwxrwxr-x  2 tecmint tecmint 4096 Aug 29 2020 .cassandra
drwx----- 13 tecmint tecmint 4096 Jan 23 20:15 .config
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Desktop
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Documents
drwxr-xr-x  2 tecmint tecmint 4096 Jan 23 19:41 Downloads
drwx-----  3 tecmint tecmint 4096 Jan 23 16:59 .gnupg
drwx-----  3 tecmint tecmint 4096 Nov 21 16:07 .local
drwx-----  5 tecmint tecmint 4096 Jan 23 17:05 .mozilla
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Music
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Pictures
-rw-r--r--  1 tecmint tecmint  807 Aug 29 2020 .profile
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Public
-rw-rw-r--  1 tecmint tecmint   0 Mar  3 10:26 reports.txt
-rw-rw-r--  1 tecmint tecmint   0 Mar  3 10:25 sales.txt
drwx-----  2 tecmint tecmint 4096 Nov 21 16:09 .ssh
-rw-r--r--  1 tecmint tecmint   0 Aug 29 2020 .sudo_as_admin_successful
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Templates
drwxr-xr-x  2 tecmint tecmint 4096 Nov 21 16:07 Videos
-rw-rw-r--  1 tecmint tecmint  165 Jan 23 18:51 .wget-hsts
drwxrwxr-x  4 tecmint tecmint 4096 Jan 23 19:47 .wine

```

List Hidden Files in Linux

4. touch Command

The touch command is used for creating simple files on a Linux system. To create a file, use the syntax:

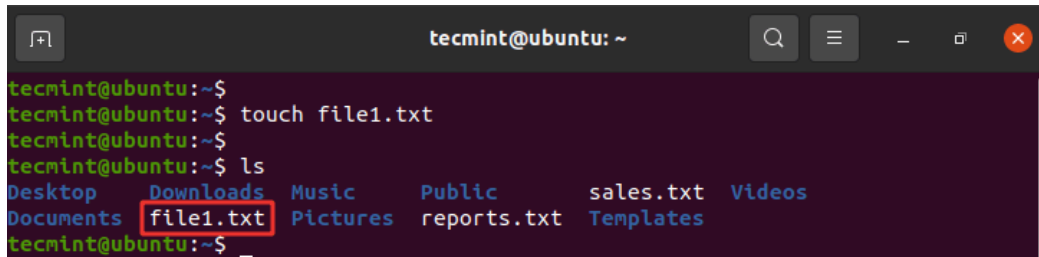
```
$ touch filename
```

For example, to create a **file1.txt** file, run the command:

```
$ touch file1.txt
```

To confirm the creation of the file, invoke the **ls command**.

```
$ ls
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ touch file1.txt
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  Videos
Documents file1.txt  Pictures   reports.txt Templates
tecmint@ubuntu:~$

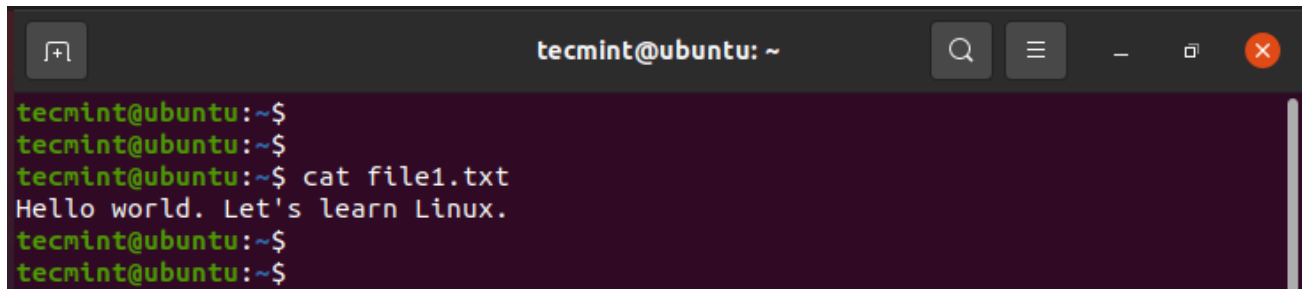
```

Create Empty File in Linux

5. cat Command

To view the contents of a file, use the cat command as follows:

```
$ cat filename
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$
tecmint@ubuntu:~$ cat file1.txt
Hello world. Let's learn Linux.
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

View Contents of Files

6. mv Command

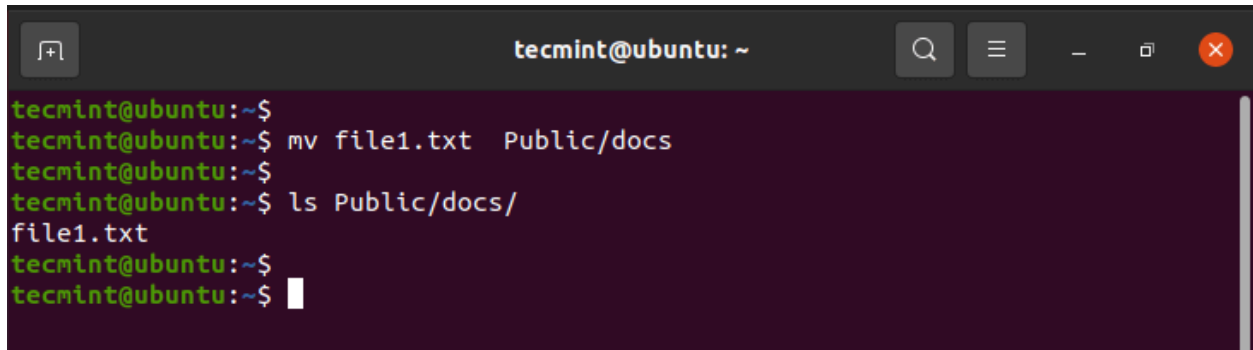
The **mv command** is quite a versatile command. Depending on how it is used, it can rename a file or move it from one location to another.

To move the file, use the syntax below:

```
$ mv filename /path/to/destination/
```

For example, to move a file from the current directory to the Public/docs directory, run the command:

```
$ mv file1.txt Public/docs
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ mv file1.txt Public/docs
tecmint@ubuntu:~$
tecmint@ubuntu:~$ ls Public/docs/
file1.txt
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

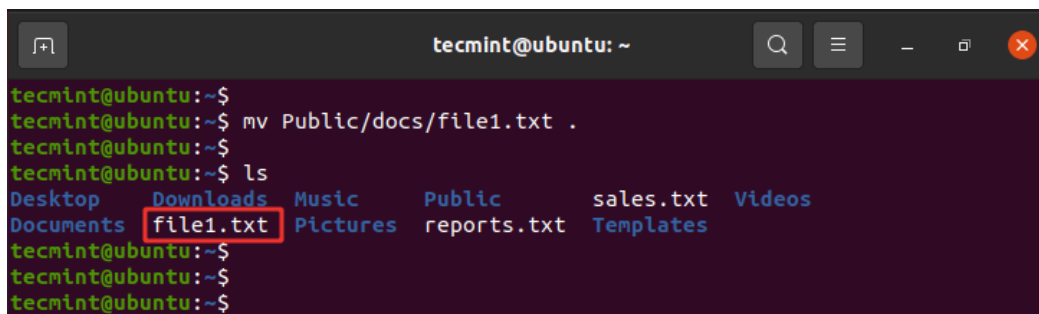
Move Files in Linux

Alternatively, you can move a file from a different location to your current directory using the syntax shown. Take note of the period sign at the end of the command. This implies this location'.

```
$ mv /path/to/file .
```

We are now going to do the reverse. We will copy the file from the Public/docs path to the current directory as shown.

```
$ mv Public/docs/file1.txt .
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ mv Public/docs/file1.txt .
tecmint@ubuntu:~$
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  Videos
Documents file1.txt  Pictures   reports.txt Templates
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

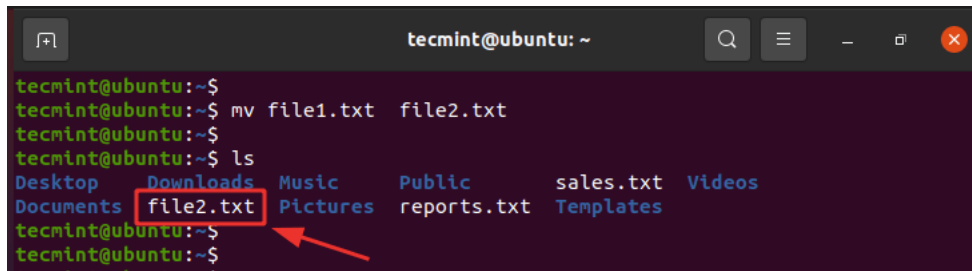
Move Files from Location in Linux

To rename a file, use the syntax shown. The command removes the original file name and assigns the second argument as the new file name.

```
$ mv filename1 filename2
```

For example, to rename file1.txt to file2.txt run the command:

```
$ mv file1.txt file2.txt
```



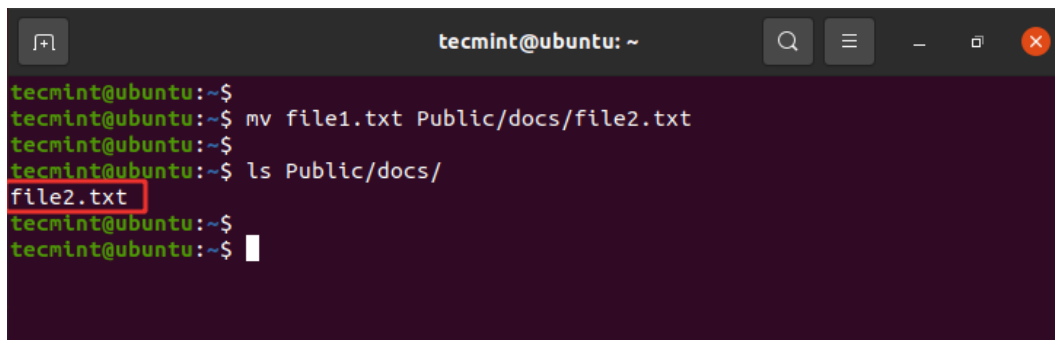
```
tecmin@ubuntu: ~
tecmin@ubuntu:~$ mv file1.txt file2.txt
tecmin@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  Videos
Documents file2.txt  Pictures   reports.txt Templates
```

Rename Files in Linux

Additionally, you can move and rename the file at the same time by specifying the destination folder and a different file name.

For example to move **file1.txt** to the location **Public/docs** and rename it **file2.txt** run the command:

```
$ mv file1.txt Public/docs/file2.txt
```



```
tecmin@ubuntu: ~
tecmin@ubuntu:~$ mv file1.txt Public/docs/file2.txt
tecmin@ubuntu:~$ ls Public/docs/
file2.txt
```

Move and Rename Files in Linux

7. cp Command

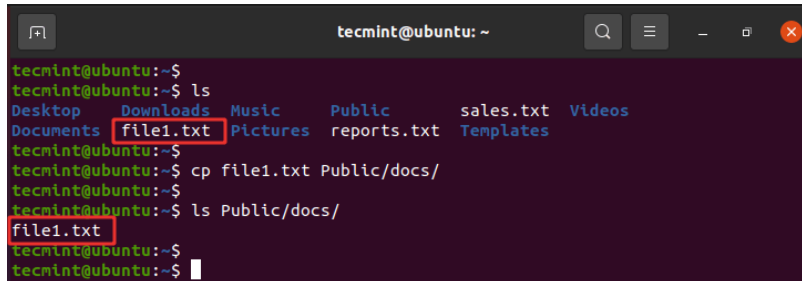
The cp command, short for copy, copies a file from one file location to another. Unlike the move command, the **cp** command retains the original file in its current location and makes a duplicate copy in a different directory.

The syntax for copying a file is shown below.

```
$ cp /file/path /destination/path
```

For example, to copy the file **file1.txt** from the current directory to the **Public/docs/** directory, issue the command:

```
$ cp file1.txt Public/docs/
```



```

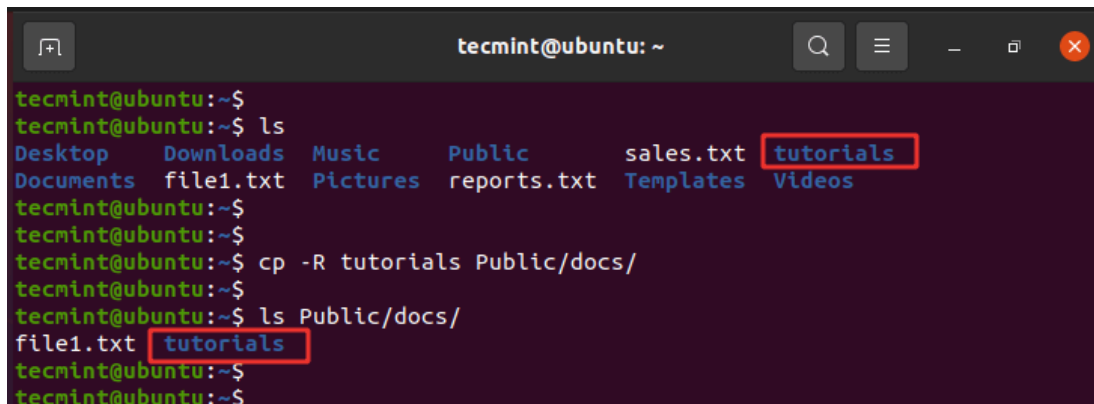
tecmint@ubuntu: ~
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  Videos
Documents file1.txt  Pictures   reports.txt Templates
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ cp file1.txt Public/docs/
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ ls Public/docs/
file1.txt
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ 

```

Copy Files in Linux

To copy a directory, use the **-R** option for recursively copying the directory including all its contents. We have created another directory called **tutorials**. To copy this directory alongside its contents to the **Public/docs/** path, run the command:

```
$ cp -R tutorials Public/docs/
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music      Public      sales.txt  tutorials
Documents file1.txt  Pictures   reports.txt Templates  Videos
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ cp -R tutorials Public/docs/
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ ls Public/docs/
file1.txt  tutorials
tecmint@ubuntu:~$ 
tecmint@ubuntu:~$ 

```

Copy Directory in Linux

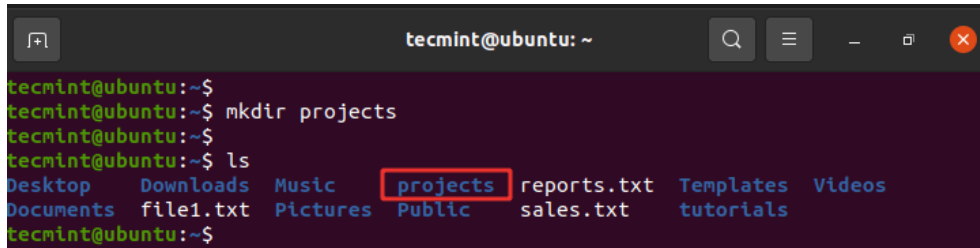
8. mkdir Command

You might have wondered how we created the **tutorials** directory. Well, it's pretty simple. To create a new directory use the **mkdir** (make directory) command as follows:

```
$ mkdir directory_name
```


Let's create another directory called **projects** as shown:

```
$ mkdir projects
```

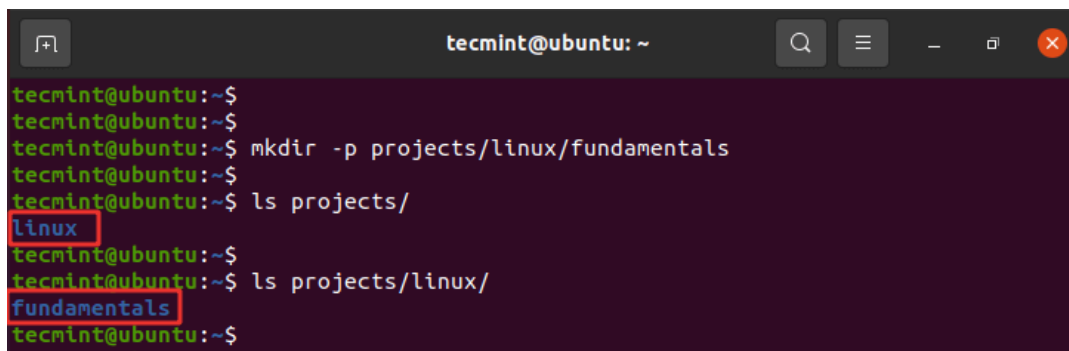


```
tecmin@ubuntu: ~
tecmin@ubuntu:~$ mkdir projects
tecmin@ubuntu:~$ ls
Desktop  Downloads  Music  projects  reports.txt  Templates  Videos
Documents  file1.txt  Pictures  Public  sales.txt  tutorials
```

Create Directory in Linux

To create a directory within another directory use the **-p** flag. The command below creates the **fundamentals** directory inside the **linux** directory within the parent directory which is the **projects** directory.

```
$ mkdir -p projects/linux/fundamentals
```



```
tecmin@ubuntu: ~
tecmin@ubuntu:~$ mkdir -p projects/linux/fundamentals
tecmin@ubuntu:~$ ls projects/
linux
tecmin@ubuntu:~$ ls projects/linux/
fundamentals
```

Create Directory in Linux

9. rmdir Command

The **rmdir** command deletes an empty directory. For example, to delete or remove the **tutorials** directory, run the command:

```
$ rmdir tutorials
```

```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music  projects  reports.txt  Templates  Videos
Documents file1.txt  Pictures Public  sales.txt  tutorials
tecmint@ubuntu:~$ rmdir tutorials
tecmint@ubuntu:~$ ls
Desktop  Downloads  Music  projects  reports.txt  Templates
Documents file1.txt  Pictures Public  sales.txt  Videos
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

Delete Empty Directory in Linux

If you try to remove a non-empty directory, you will get an error message as shown.

```
$ rmdir projects
```

```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ rmdir projects
rmdir: failed to remove 'projects': Directory not empty
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

Delete Directory in

Linux

10. rm Command

The **rm** (remove) command is used to delete a file. The syntax is quite straightforward:

```
$ rm filename
```

For example, to delete the **file1.txt** file, run the command:

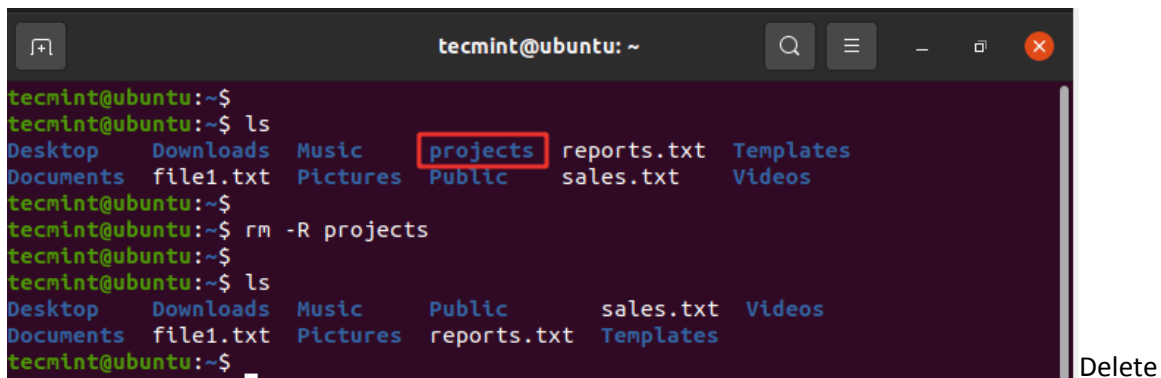
```
$ rm file1.txt
```

Additionally, you can remove or delete a directory recursively using the `-R` option. This could either be an empty or a non-empty directory.

```
$ rm -R directory_name
```

For example, to delete the **projects** directory, run the command:

```
$ rm -R projects
```



A terminal window titled 'tecmin@ubuntu: ~' showing the following commands and output:

```
tecmin@ubuntu:~$
tecmin@ubuntu:~$ ls
Desktop  Downloads  Music  projects  reports.txt  Templates
Documents file1.txt  Pictures Public    sales.txt    Videos
tecmin@ubuntu:~$
tecmin@ubuntu:~$ rm -R projects
tecmin@ubuntu:~$
tecmin@ubuntu:~$ ls
Desktop  Downloads  Music  Public    sales.txt  Videos
Documents file1.txt  Pictures reports.txt Templates
tecmin@ubuntu:~$
```

The word 'Delete' is written to the right of the terminal output.

Directory Recursively in Linux

11. find and locate Commands

Sometimes, you may want to search the location of a particular file. You can easily do this using either the **find** or **locate** commands.

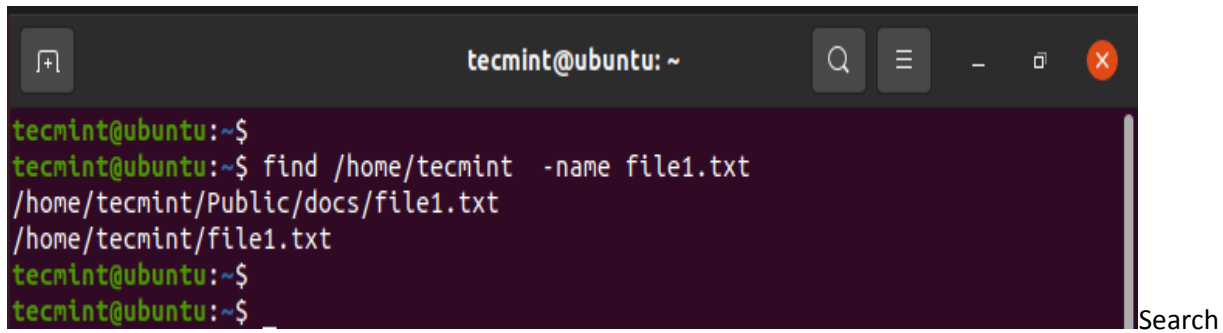
The **find** command searches for a file in a particular location and takes two arguments: the search path or directory and the file to be searched.

The syntax is as shown

```
$ find /path/to/search -name filename
```

For example, to search for a file called **file1.txt** in the home directory, run:

```
$ find /home/tecmin -name file1.txt
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ find /home/tecmint -name file1.txt
/home/tecmint/Public/docs/file1.txt
/home/tecmint/file1.txt
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

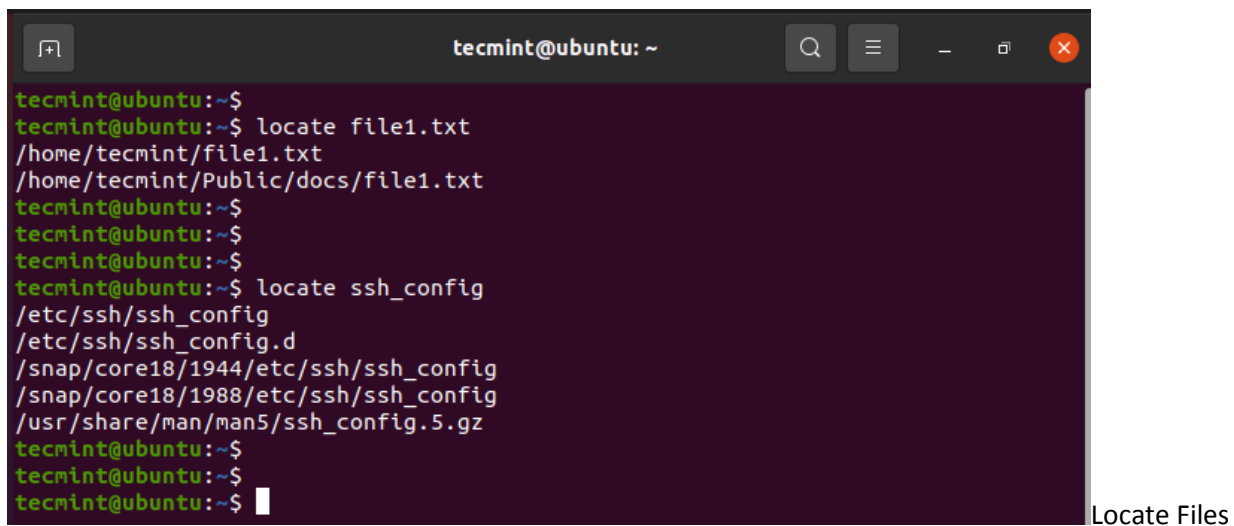
Files in Linux

The **locate** command, just like the **find** command, plays the same role of searching files but only takes one argument as shown.

```
$ locate filename
```

For example;

```
$ locate file1.txt
```



```

tecmint@ubuntu: ~
tecmint@ubuntu:~$ locate file1.txt
/home/tecmint/file1.txt
/home/tecmint/Public/docs/file1.txt
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$ locate ssh_config
/etc/ssh/ssh_config
/etc/ssh/ssh_config.d
/snap/core18/1944/etc/ssh/ssh_config
/snap/core18/1988/etc/ssh/ssh_config
/usr/share/man/man5/ssh_config.5.gz
tecmint@ubuntu:~$
tecmint@ubuntu:~$
tecmint@ubuntu:~$

```

in Linux

The **locate** command searches using a database of all the possible files and directories in the system.

NOTE: The **locate** command is much faster than the **find** command. However, the **find** command is much more powerful and works in situations where **locate** does not produce the desired results.

File & Directory types in Linux

Introduction

When navigating the Linux file system you are sure to encounter different file types. The most used and obvious file types are regular files and directories. However, the Linux operating system has more to offer in terms of file types as it also includes another 5 file types. This short article will help you to recognize all the 7 different file types within the Linux operating system.

Identifying Linux File types

There is only 1 command you need to know, which will help you to identify and categorize all the seven different file types found on the Linux system.

```
$ ls -ld <file name>
```

Here is an example output of the above command.

```
$ ls -ld /etc/services
-rw-r--r-- 1 root root 19281 Feb 14 2012 /etc/services
```

ls command will show the file type as an encoded symbol found as the first character of the file permission part. In this case it is “-“, which means “regular file”. It is important to point out that Linux file types are not to be mistaken with file extensions. Let us have a look at a short summary of all the seven different types of Linux file types and **ls** command identifiers:

1. **-** : regular file
2. **d** : directory
3. **c** : character device file
4. **b** : block device file
5. **s** : local socket file
6. **p** : named pipe
7. **l** : symbolic link

Regular file

The regular file is a most common file type found on the Linux system. It governs all different files such as text files, images, binary files, shared libraries, etc. You can create a regular file with the **touch** command:

```
$ touch linuxcareer.com
$ ls -ld linuxcareer.com
-rw-rw-r-- 1 lubos lubos 0 Jan 10 12:52 linuxcareer.com
```

The first character of the **ls** command, in this case “-“, denotes the identification code for the regular file. To remove a regular file you can use the **rm** command:

```
$ rm linuxcareer.com
$
```

Directory

Directory is second most common file type found in Linux. Directory can be created with the **mkdir** command:

```
$ mkdir FileTypes
$ ls -ld FileTypes/
drwxrwxr-x 2 lubos lubos 4096 Jan 10 13:14 FileTypes/
```

As explained earlier, directory can be identified by “d” symbol from the **ls** command output. To remove empty directory use the **rmdir** command.

```
$ rmdir FileTypes
```

When trying to remove directory with the **rmdir** command, which contains additional files you will get an error message:

```
rmdir: failed to remove `FileTypes/': Directory not empty
```

In this case you need to use a command:

```
$ rm -r FileTypes/
```

Character device

Character and block device files allow users and programs to communicate with hardware peripheral devices. For example:

```
$ ls -ld /dev/vmmon
crw----- 1 root root 10, 165 Jan  4 10:13 /dev/vmmon
```

In this case the character device is the vmware module device.

The Boot Process

Below are the basic stages of the boot process for an x86 system:

1. The system BIOS checks the system and launches the first stage boot loader on the MBR of the primary hard disk.

2. The first stage boot loader loads itself into memory and launches the second stage boot loader from the `/boot/` partition.
3. The second stage boot loader loads the kernel into memory, which in turn loads any necessary modules and mounts the root partition read-only.
4. The kernel transfers control of the boot process to the `/sbin/init` program.
5. The `/sbin/init` program loads all services and user-space tools, and mounts all partitions listed in `/etc/fstab`.
6. The user is presented with a login screen for the freshly booted Linux system.

Because configuration of the boot process is more common than the customization of the shutdown process, the remainder of this chapter discusses in detail how the boot process works and how it can be customized to suite specific needs.

Booting and Shutting Down

Introduction

One of the most powerful aspects of Linux concerns its open method of starting and stopping the operating system, where it loads specified programs using their particular configurations, permits you to change those configurations to control the boot process, and shuts down in a graceful and organized way.

Beyond the question of controlling the boot or shutdown process, the open nature of Linux makes it much easier to determine the exact source of most problems associated with starting up or shutting down your system. A basic understanding of this process is quite beneficial to everybody who uses a Linux system.

A lot of Linux systems use lilo, the LINUX LOader for booting operating systems. We will only discuss GRUB, however, which is easier to use and more flexible. Should you need information about lilo, refer to the man pages and HOWTOs. Both systems support dual boot installations, we refer to the HOWTOs on this subject for practical examples and background information.

The boot process

When an x86 computer is booted, the processor looks at the end of the system memory for the BIOS (Basic Input/Output System) and runs it. The BIOS program is written into permanent read-only memory and is always available for use. The BIOS provides the lowest level interface to peripheral devices and controls the first step of the boot process.

The BIOS tests the system, looks for and checks peripherals, and then looks for a drive to use to boot the system. Usually it checks the floppy drive (or CD-ROM drive on many newer systems) for bootable media, if present, and then it looks to the hard drive. The order of the drives used for booting is usually controlled by a particular BIOS setting on the system. Once Linux is installed on the hard drive of a system, the BIOS looks for a Master Boot Record (MBR) starting at the first sector on the first hard drive, loads its contents into memory, then passes control to it.

This MBR contains instructions on how to load the GRUB (or LILO) boot-loader, using a pre-selected operating system. The MBR then loads the boot-loader, which takes over the process (if the boot-loader is installed in the MBR). In the default Red Hat Linux configuration, GRUB uses the settings in the MBR to display boot options in a menu. Once GRUB has received the correct

instructions for the operating system to start, either from its command line or configuration file, it finds the necessary boot file and hands off control of the machine to that operating system.

GRUB features

This boot method is called *direct loading* because instructions are used to directly load the operating system, with no intermediary code between the boot-loaders and the operating system's main files (such as the kernel). The boot process used by other operating systems may differ slightly from the above, however. For example, Microsoft's DOS and Windows operating systems completely overwrite anything on the MBR when they are installed without incorporating any of the current MBR's configuration. This destroys any other information stored in the MBR by other operating systems, such as Linux. The Microsoft operating systems, as well as various other proprietary operating systems, are loaded using a chain loading boot method. With this method, the MBR points to the first sector of the partition holding the operating system, where it finds the special files necessary to actually boot that operating system.

GRUB supports both boot methods, allowing you to use it with almost any operating system, most popular file systems, and almost any hard disk your BIOS can recognize.

GRUB contains a number of other features; the most important include:

- GRUB provides a true command-based, pre-OS environment on x86 machines to allow maximum flexibility in loading operating systems with certain options or gathering information about the system.
- GRUB supports Logical Block Addressing (LBA) mode, needed to access many IDE and all SCSI hard disks. Before LBA, hard drives could encounter a 1024-cylinder limit, where the BIOS could not find a file after that point.
- GRUB's configuration file is read from the disk every time the system boots, preventing you from having to write over the MBR every time you change the boot options.

Init

The kernel, once it is loaded, finds init in sbin and executes it.

When init starts, it becomes the parent or grandparent of all of the processes that start up automatically on your Linux system. The first thing init does, is reading its initialization file, /etc/inittab. This instructs init to read an initial configuration script for the environment, which sets the path, starts swapping, checks the file systems, and so on. Basically, this step takes care of everything that your system needs to have done at system initialization: setting the clock, initializing serial ports and so forth.

Then init continues to read the /etc/inittab file, which describes how the system should be set up in each run level and sets the default *run level*. A run level is a configuration of processes. All UNIX-like systems can be run in different process configurations, such as the single user mode, which is referred to as run level 1 or run level S (or s). In this mode, only the system administrator can connect to the system. It is used to perform maintenance tasks without risks of damaging the system or user data. Naturally, in this configuration we don't need to offer user services, so they will all be disabled. Another run level is the reboot run level, or run level 6, which shuts down all running services according to the appropriate procedures and then restarts the system.

After having determined the default run level for your system, `init` starts all of the background processes necessary for the system to run by looking in the appropriate `rc` directory for that run level. `init` runs each of the kill scripts (their file names start with a `K`) with a stop parameter. It then runs all of the start scripts (their file names start with an `S`) in the appropriate run level directory so that all services and applications are started correctly. In fact, you can execute these same scripts manually after the system is finished booting with a command

like **`/etc/init.d/httpd stop`** or **`service httpd stop`** logged in as *root*, in this case stopping the web server.

After `init` has progressed through the run levels to get to the default run level, the `/etc/inittab` script forks a `getty` process for each virtual console (login prompt in text mode). `getty` opens `tty` lines, sets their modes, prints the login prompt, gets the user's name, and then initiates a login process for that user. This allows users to authenticate themselves to the system and use it. By default, most systems offer 6 virtual consoles, but as you can see from the `inittab` file, this is configurable.

`/etc/inittab` can also tell **`init`** how it should handle a user pressing `Ctrl+Alt+Delete` at the console. As the system should be properly shut down and restarted rather than immediately power-cycled, `init` is told to execute the command `/sbin/shutdown -t3 -r now`, for instance, when a user hits those keys. In addition, `/etc/inittab` states what `init` should do in case of power failures, if your system has a UPS unit attached to it.

On most RPM-based systems the graphical login screen is started in run level 5, where `/etc/inittab` runs a script called `/etc/X11/prefdm`. The `prefdm` script runs the preferred X display manager, based on the contents of the `/etc/sysconfig/desktop` directory. This is typically `gdm` if you run GNOME or `kdm` if you run KDE, but they can be mixed, and there's also the `xdm` that comes with a standard X installation.

Shutdown

UNIX was not made to be shut down, but if you really must, use the `shutdown` command. After completing the shutdown procedure, the `-h` option will halt the system, while `-r` will reboot it.

The `reboot` and `halt` commands are now able to invoke `shutdown` if run when the system is in run levels 1-5, and thus ensure proper shutdown of the system, but it is a bad habit to get into, as not all UNIX/Linux versions have this feature.

If your computer does not power itself down, you should not turn off the computer until you see a message indicating that the system is halted or finished shutting down, in order to give the system the time to unmount all partitions. Being impatient may cause data loss.