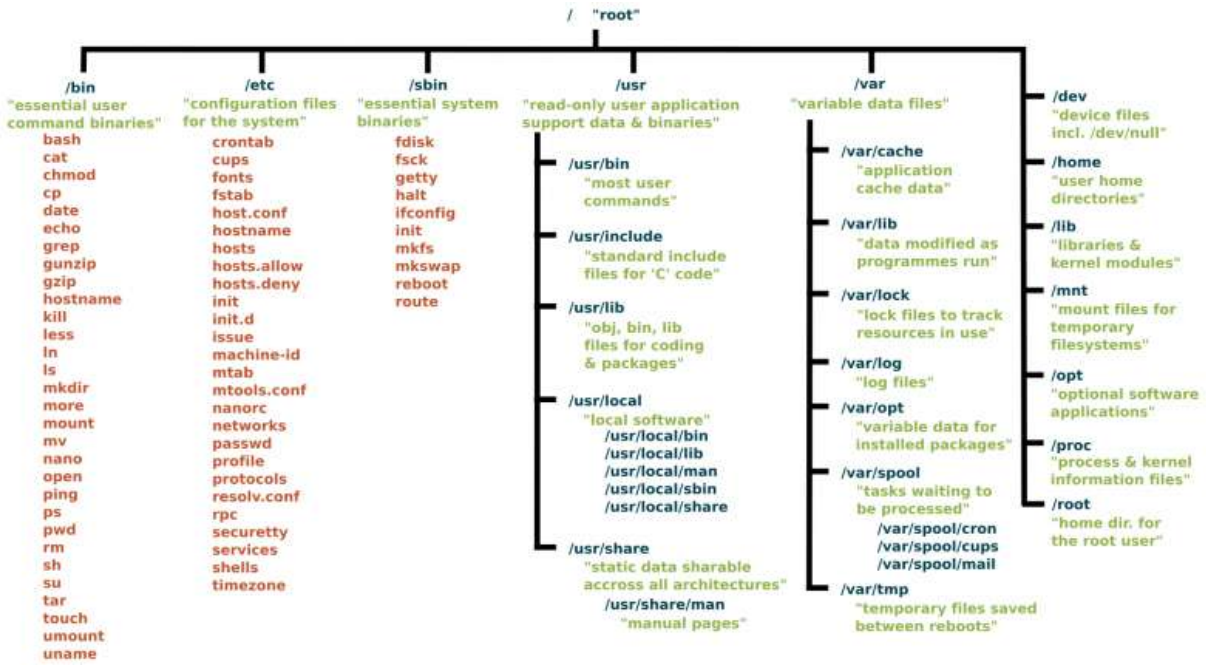
Linux file system

A simple description of the Linux system, is this:

"On Linux, everything is a file. if something is not a file, it is a process."

This statement is true because there are special files that are more than just files (named pipes and sockets, for instance). A Linux system, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

In order to manage all those files in an orderly fashion, we like to think of them in an ordered tree-like structure on the hard disk. Beginning with the root, the large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out that this is not a fully accurate image.



Beside regular files like text files, picture files, executable files, and so on, there are some exceptions:

- Directories: files that are lists of other files

- Links: a system to make a file or directory visible in multiple parts of the file tree.

- Special files: interface to a device driver (mostly in /dev)

- (Domain) sockets: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.

- Named pipes: similar to sockets and form a way for different processes to communicate with each other, without using network socket semantics.

- Block devices: commonly represent hardware such as disk drives.

In a long list these file types are marked with the following symbols:

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe
b	Block device

For users, it might be enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures.

Every partition has its own file system. By imagining all those file systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an "inode" (index node), a kind of structure containing information about the actual data that makes up the file. Every partition has its own set of inodes. Inodes are unique only on the partition level, so that in systems with multiple partitions, files with the same inode number can exist.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

When a new file is created, it gets a free inode. The inode contains the following information:

- Owner and group owner of the file.

- File type (regular, directory, ...)

- Permissions for the file

- Date and time of creation, last read and change.

- Date and time this information has been changed in the inode.

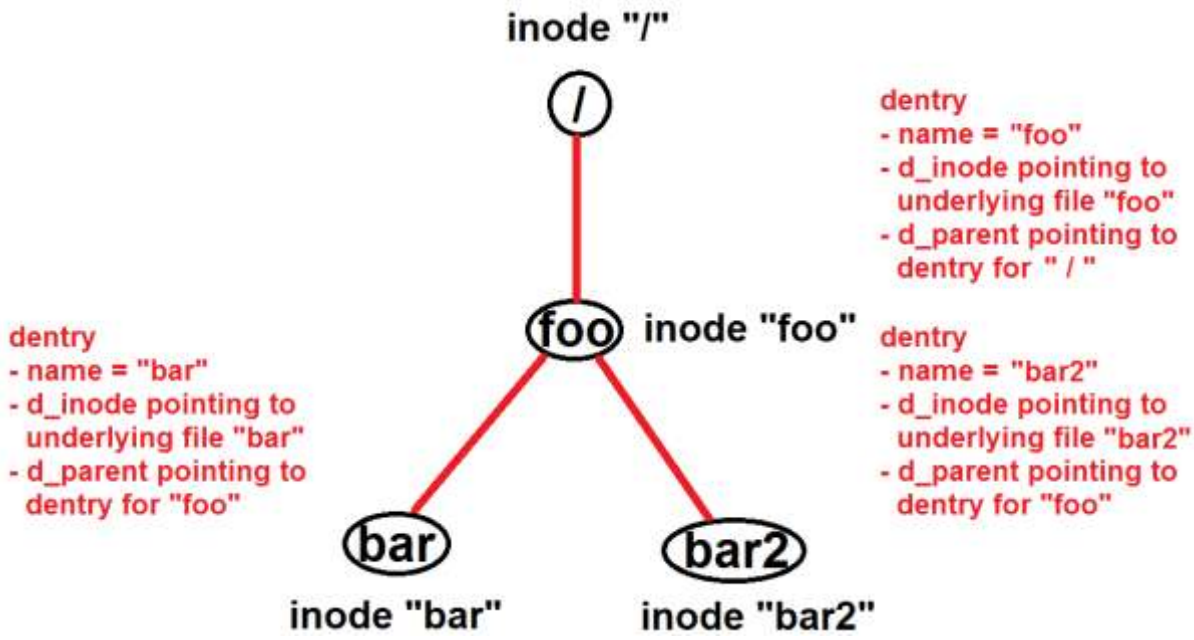
- Number of links to this file.

- File size

- An address defining the actual location of the file data.

The information not included in an inode, is the file name and directory. These are stored in special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. The inode numbers can be displayed using ls with the -i option. Inodes have their own separate space on the disk.

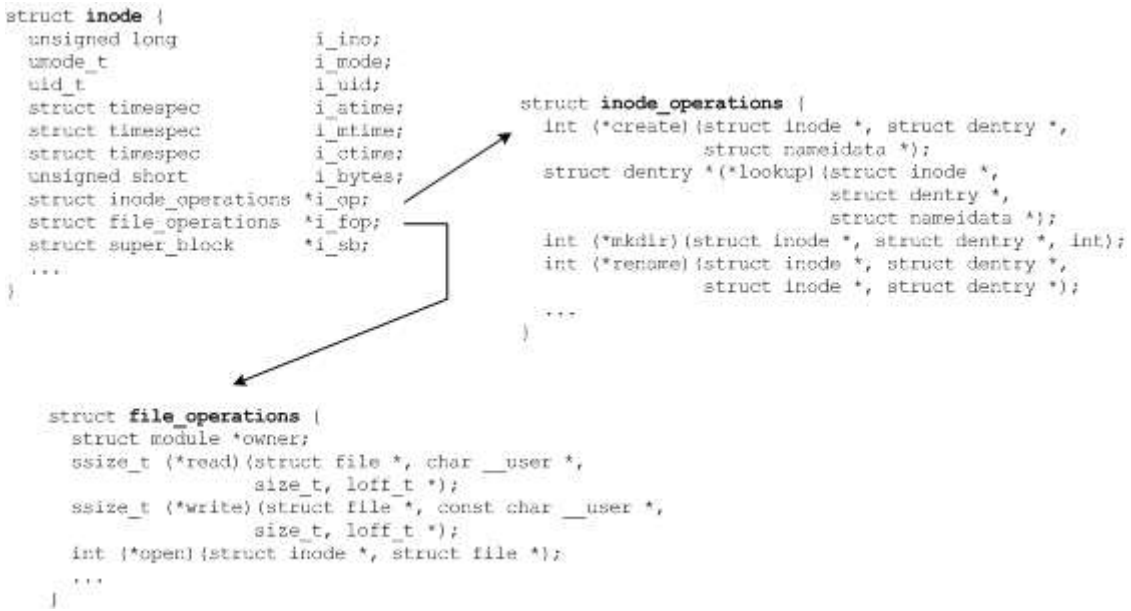
The individual file systems provide methods for translating a filename into a unique inode identifier and then to an inode reference. In the VFS (Virtual File Systems), the directories are represented in a structure called dentry. The dentry is a C structure with a string name (d_name), a pointer to an inode (d_inode) and a pointer to the parent dentry (d_parent).



A Tree such as shown above is represented by 4 inodes, one each for "foo", "bar", "bar2" and the "/" (root) as well as 3 dentries: one linking "bar" to "foo", one linking "bar2" to "foo", and one linking "foo" to the "root". The first of those dentries, for example, has a name of "bar", a d_inode pointing to the underlying file "bar", and a d_parent pointing to the dentry for "foo" (which in turn would have a d_parent pointing to the dentry for the root). The root dentry has a d_parent that points to itself.

Note that the mapping from dentries to inodes given by d_inode is in general a many-to-one mapping. A single file may be pointed to by multiple paths in the same filesystem (called "hard links").

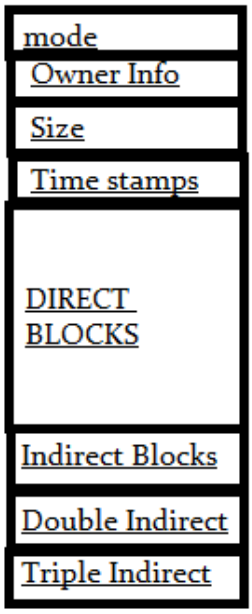
The inode and dentry structures are defined in ./linux/include/linux/fs.h.



Note in particular the inode_operations and file_operations. Each of these structures refers to the individual operations that may be performed on the inode. For example, inode_operations define those operations that operate directly on the inode and file_operations refer to those methods related to files and directories (the standard system calls). The most-recently used inodes and dentries are kept in the inode and directory cache respectively. For each inode in the inode cache there is a corresponding dentry in the directory cache.

Inode Structure of a File

Now let's see how the structure of an inode of a file looks like.



- Mode: This keeps information about two things, one is the permission information, the other is the type of inode, for example an inode can be of a file, directory or a block device etc.

- Owner Info: Access details like owner of the file, group of the file etc.

- Size: This location stores the size of the file in terms of bytes.

- Time Stamps: Stores the inode creation time, modification time, etc.

Now comes the important thing to understand about how a file is saved in a partition with the help of an inode.

- Block Size: Whenever a partition is formatted with a file system. It normally gets formatted with a default block size. Now block size is the size of chunks in which data will be spread. So if the block size is 4K, then for a file of 15K it will take 4 blocks (because 4K*4 16), and technically speaking you waste 1 K.

- Direct Block Pointers: In an ext2 file system an inode consists of only 15 block pointers. The first 12 block pointers are called as Direct Block pointers. Which means that these pointers point to the address of the blocks containing the data of the file. 12 Block pointers can point to 12 data blocks. So in total the Direct Block pointers can address only 48K(12 * 4K) of data. Which means if the file is only of 48K or below in size, then inode itself can address all the blocks containing the data of the file.

Now What if the file size is above 48K?

- Indirect Block Pointers: whenever the size of the data goes above 48k (by considering the block size as 4k), the 13th pointer in the inode will point to the very next block after the data (adjacent block after 48k of data), which inturn will point to the next block address where data is to be copied.

Now as we have took our block size as 4K, the indirect block pointer, can point to 1024 blocks containing data (by taking the size of a block pointer as 4bytes, one 4K block can point to 1024 blocks because 4 bytes * 1024 = 4K).

which means an indirect block pointer can address, upto 4MB of data(4bytes of block pointer in 4K block, can point and address 1024 number of 4K blocks which makes the data size of 4M)

- Double indirect Block Pointers: Now if the size of the file is above 4MB + 48K then the inode will start using Double Indirect Block Pointers, to address data blocks. Double Indirect Block pointer in an inode will point to the block that comes just after 4M + 48K data, which intern will point to the blocks where the data is stored. Double Indirect block pointer also is inside a 4K block as every blocks are 4K, Now block pointers are 4 bytes in size, as mentioned previously, so Double indirect block pointer

can address 1024 Indirect Block pointers (which means 1024 * 4M =4G). So with the help of a double indirect Block Pointer the size of the data can go upto 4G.

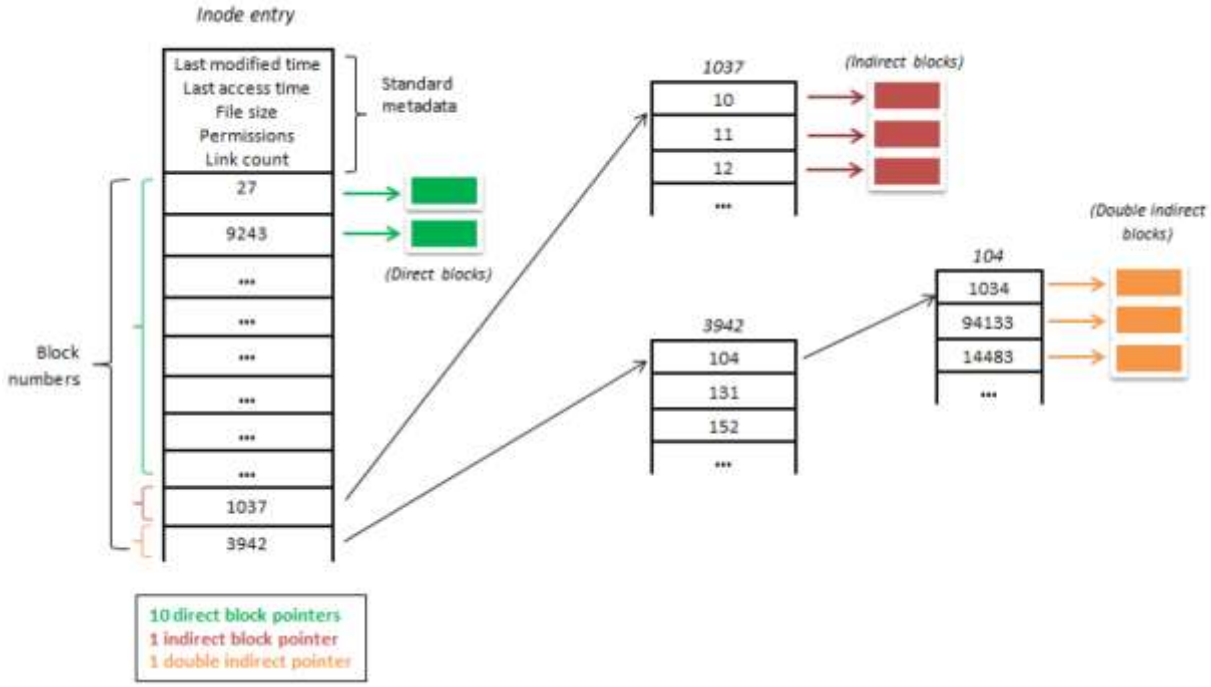
- Triple Indirect Block Pointers: Now this triple Indirect Block Pointers can address upto 4G * 1024 = 4TB, of file size. The fifteenth block pointer in the inode will point to the block just after the 4G of data, which intern will point to 1024 Double Indirect Block Pointers.

So after the 12 direct block pointers, 13th block pointer in inode is for Indirect block pointers, and 14th block pointer is for double indirect block pointers, and 15th block pointer is for triple indirect block pointers.

This is the main reason why there are limits to the full size of a single file that you can have in a file system.

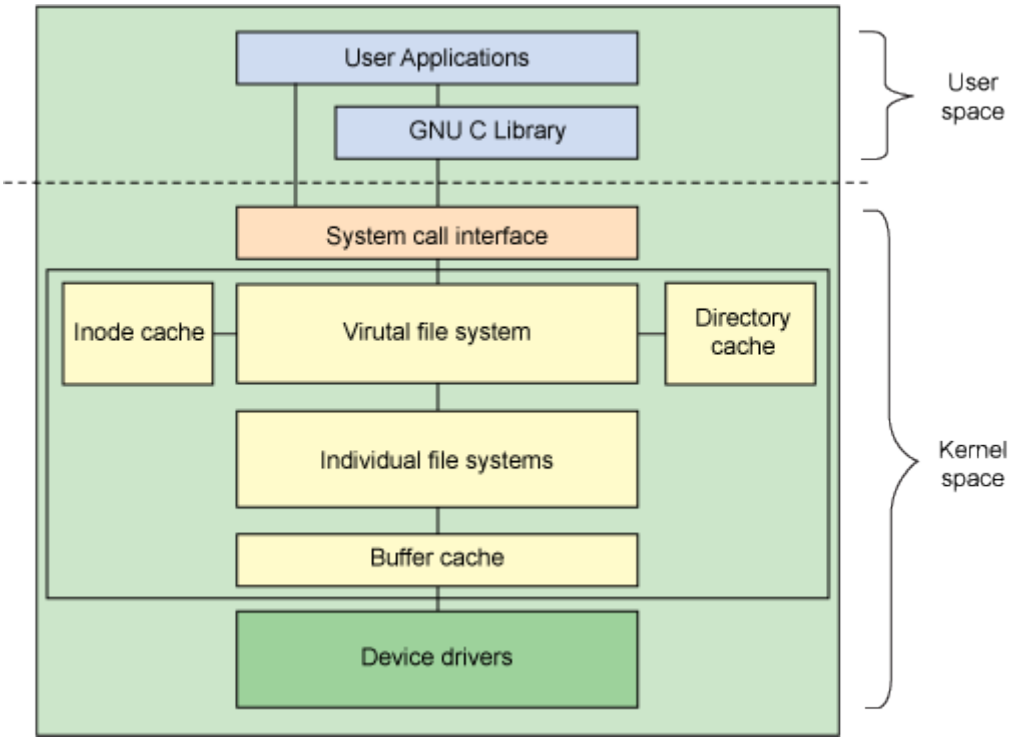
Now an interesting fact to understand is that the total no of inodes are created at the time of creating a file system. Which means there is an upper limit in the number of inodes you can have in a file system. Now after that limit has reached you will not be able to create any more files on the file system, even if you have space left on the partition.

The principle of indirect block pointers is shown below:



High-level architecture

While the majority of the file system code exists in the kernel (except for user-space file systems), the architecture of the linux file system shown below shows the relationships between the major file system- related components in both user space and the kernel.



User space contains the applications (for this example, the user of the file system) and the GNU C Library (glibc), which provides the user interface for the file system calls (open, read, write, close). The system call interface acts as a switch, funneling system calls from user space to the appropriate endpoints in kernel space.

The VFS is the primary interface to the underlying file systems. This component exports a set of interfaces and then abstracts them to the individual file systems (ext2, ext3, etc.), which may behave very differently from one another. Two caches exist for file system objects (inodes and dentries). Each provides a pool of recently-used file system objects.

Each individual file system implementation, such as ext2, JFS, and so on, exports a common set of interfaces that is used (and expected) by the VFS.

The buffer cache buffers requests between the file systems and the block devices that they manipulate. For example, read and write requests to the underlying device drivers migrate through the buffer cache. This allows the requests to be cached there for faster access (rather than going back out to the physical device). The buffer cache is managed as a set of least recently used (LRU) lists. Note that you can use the sync command to flush the buffer cache out to the storage media (force all unwritten data out to the device drivers and, subsequently, to the storage device).

Major structures

Linux views all file systems from the perspective of a common set of objects. These objects are the superblock, inode, dentry, and file. At the root of each file system is the superblock, which describes and maintains state for the file system.

Every object that is managed within a file system (file or directory) is represented in Linux as an inode. The inode contains all the metadata to manage objects in the file system (including the operations that are possible on it). Another set of structures, called dentries, is used to translate between names and inodes, for which a directory cache exists to keep the most-recently used around. The dentry also maintains relationships between directories and files for traversing file systems. Finally, a VFS file represents an open file (keeps state for the open file such as the write offset, and so on).

Another set of structures, called dentries, is used to translate between names and inodes, for which a directory cache exists to keep the most-recently used around. The dentry also maintains relationships between directories and files for traversing file systems. Finally, a VFS file represents an open file (keeps state for the open file such as the write offset, and so on).

Virtual file system

The VFS acts as the root level of the file-system interface. The VFS keeps track of the currentlysupported file systems, as well as those file systems that are currently mounted. File systems can be dynamically added or removed from Linux using a set of registration functions. The kernel keeps a list of currently-supported file systems, which can be viewed from user space through the /proc file system. This virtual file also shows the devices currently associated with the file systems. To add a new file system to Linux, register_filesystem is called. This takes a single argument defining the reference to a file system structure (file_system_type), which defines the name of the file system, a set of attributes, and two superblock functions. A file system can also be unregistered. Registering a new file system places the new file system and its pertinent information onto a file_systems list. This list defines the file systems that can be supported. You can view this list by typing cat /proc/filesystems at the command line.

Superblock

The superblock is a structure that represents a file system. It includes the necessary information to manage the file system during operation. It includes the file system name (such as ext2), the size of the file system and its state, a reference to the block device, and metadata information (such as free lists and so on). The superblock is

typically stored on the storage medium but can be created in real time if one doesn't exist. One important element of the superblock is a definition of the superblock operations. The superblock structure defines the set of functions for managing inodes within the file system. For example, inodes can be allocated with alloc_inode or deleted with destroy_inode. You can read and write inodes with read_inode and write_inode or sync the file system with sync_fs. Each file system provides its own inode methods, which implement the operations and provide the common abstraction to the VFS layer.

User space / Kernel space

Memory gets divided into two distinct areas:

The division of system memory in Linux into user space and kernel space plays an important role in maintaining system stability and security.

The user space, which is a set of locations where normal user processes run (i.e everything other than the kernel). The role of the kernel is to manage applications running in this space from messing with each other, and the machine.

The kernel space, which is the location where the code of the kernel is stored, and executes under.

Processes running under the user space have access only to a limited part of memory, whereas the kernel has access to all of the memory. Processes running in user space also don't have access to the kernel space. User space processes can only access a small part of the kernel via an interface exposed by the kernel - the system calls. If a process performs a system call, a software interrupt is sent to the kernel, which then dispatches the appropriate interrupt handler and continues its work after the handler has finished.

Kernel space code has the property to run in "kernel mode", which (in typical desktop -x86- computer) is what you call code that executes under ring 0. Typically in x86 architecture, there are 4 rings of protection. Ring 0 (kernel mode), Ring 1 (may be used by virtual machine hypervisors or drivers), Ring 2 (may be used by drivers). Ring 3 is what typical applications run under. It is the least privileged ring, and applications running on it have access to a subset of the processor's instructions. Ring 0 (kernel space) is the most privileged ring, and has access to all of the machine's instructions.

User Management

User management is a critical part of maintaining a secure system. Ineffective user and privilege management often lead many systems into being compromised. Therefore, it is important to understand how you can protect your server through simple and effective user account management techniques.

Ubuntu developers made a conscientious decision to disable the administrative root account by default in all Ubuntu installations. This does not mean that the root account has been deleted or that it may not be accessed. It merely has been given a password which matches no possible encrypted value, therefore may not log in directly by itself.

Instead, users are encouraged to make use of a tool by the name of sudo to carry out system administrative duties. Sudo allows an authorized user to temporarily elevate their privileges using their own password instead of having to know the password belonging to the root account. This simple yet effective methodology provides accountability for all user actions, and gives the administrator granular control over which actions a user can perform with said privileges.

If for some reason you wish to enable the root account, simply give it a password.

By default, the initial user created by the Ubuntu installer is a member of the group "sudo" which is added to the file /etc/sudoers as an authorized sudo user. If you wish to give any other account full root access through sudo, simply add them to the sudo group.

- Adding and Deleting Users

The process for managing local users and groups is straightforward and differs very little from most other GNU/Linux operating systems. Ubuntu and other Debian based distributions encourage the use of the "adduser" package for account management.

Deleting an account does not remove their respective home folder. It is up to you whether or not you wish to delete the folder manually or keep it according to your desired retention policies.

Remember, any user added later on with the same UID/GID as the previous owner will now have access to this folder if you have not taken the necessary precautions.

You may want to change these UID/GID values to something more appropriate, such as the root account, and perhaps even relocate the folder to avoid future conflicts.

- User Profile Security

When a new user is created, the adduser utility creates a brand new home directory named /home/username. The default profile is modeled after the contents found in the directory of /etc/skel, which includes all profile basics. If your server will be home to multiple users, you should pay close attention to the user home directory

permissions to ensure confidentiality. By default, user home directories in Ubuntu are created with world read/execute permissions. This means that all users can browse and access the contents of other users home directories. This may not be suitable for your environment.

A much more efficient approach to the matter would be to modify the adduser global default permissions when creating user home folders. Simply edit the file /etc/adduser.conf and modify the DIR_MODE variable to something appropriate, so that all new home directories will receive the correct permissions.

- Password Policy

A strong password policy is one of the most important aspects of your security posture. Many successful security breaches involve simple brute force and dictionary attacks against weak passwords. If you intend to offer any form of remote access involving your local password system, make sure you adequately address minimum password complexity requirements, maximum password lifetimes, and frequent audits of your authentication systems.

Minimum Password Length: By default, Ubuntu requires a minimum password length of 6 characters, as well as some basic entropy checks.

- Password Expiration

When creating user accounts, you should make it a policy to have a minimum and maximum password age forcing users to change their passwords when they expire.