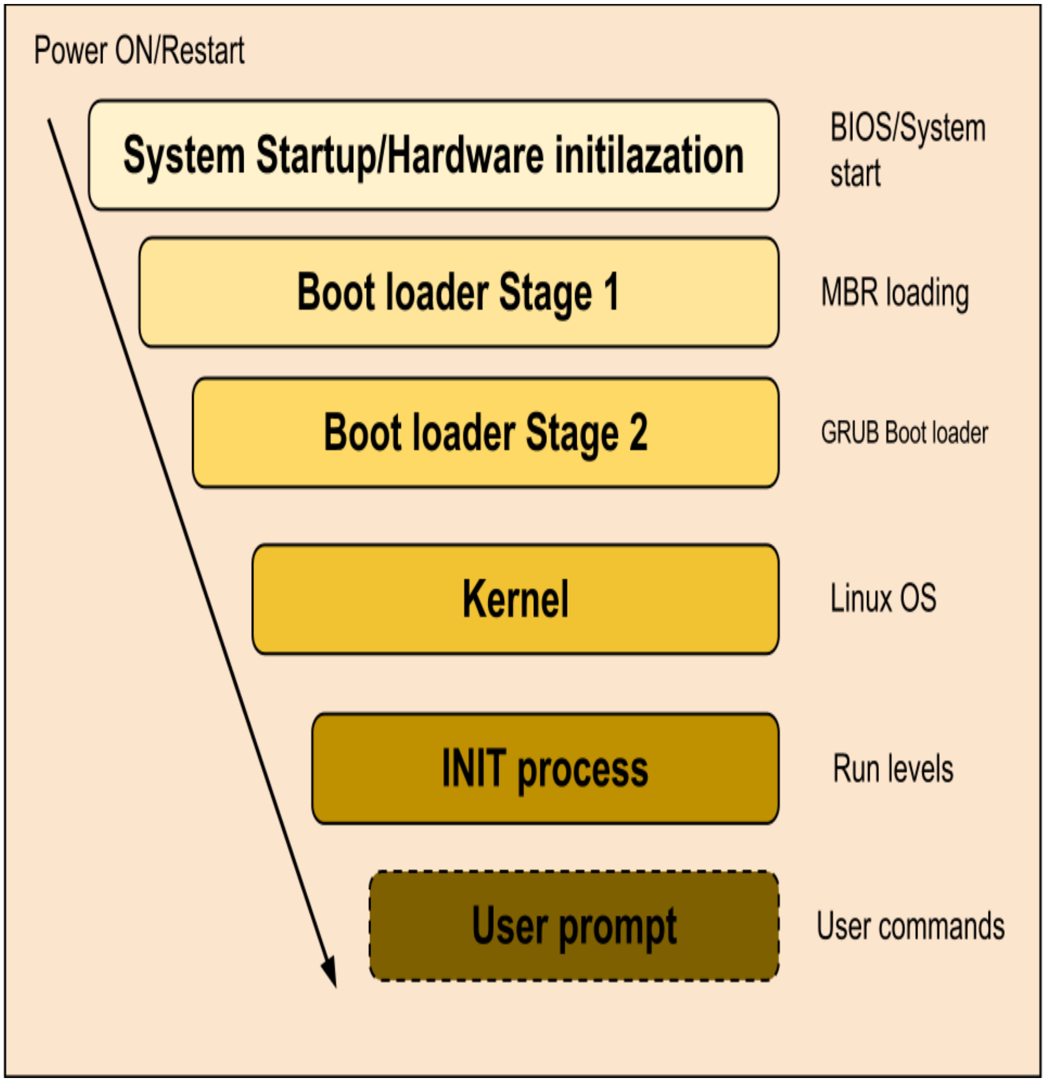
Linux Boot Process :

Knowing Linux booting process is an essential part of every Linux user/administration which will give you a clear picture of how Linux Operating system works. In this post we will see what happens when a Linux OS boots i.e. after powering on the machine to the user login prompt. Below image will give you clear idea what will happen in Linux booting process.



A quick view of booting sequence:
Power on
CPU jumps to BIOS
BIOS runs POST

Finds first bootable device
Load and execute MBR
Load OS
User prompt
This a rough idea what happens in Linux booting. Below are the detailed stages in Linux Booting process.

Stages of Booting:
1)System startup(Hardware)
2)Boot loader Stage 1 (MBR loading)
3)Boot loader Stage 2 (GRUB loader)
4)Kernel
5)INIT
6)User prompt

Stage 1: System startup

This is the first stage of booting process. When you power on/Restart your machine the power is supplied to SMPS (switched-mode power supply) which converts AC to DC. The DC power is supplied to all the devices connected to that machine such as Motherboard HDD's, CD/DVD-ROM, Mouse, keyboard etc. The most intelligent device in the computer is Processor(CPU), when supplied with power will start running its sequence operations stored in its memory. The first instruction it will run is to pass control to BIOS(Basic Input/Output System) to do POST(Power On Self Test). Once the control goes to BIOS it will take care of two things

- Run POST operation.
- Selecting first Boot device.

POST operation: POST is a processes of checking hardware availability. BIOS will have a list of all devices which are present in previous system boot. In order to check if a hardware is available for the present booting or not it will send an electric pulse to each and every device in the list that it already have. If an electrical pulse is returned from that device it will come to a conclusion the hardware is working fine and ready for use. If it does not receive a single from a particular device it will treat that device as faulty or it was removed from the system. If any new hardware is attached to the system it will do the same operation to find if its available or not. The new list will be stored in BIOS memory for next boot.
Selecting First Boot Device: Once the POST is completed BIOS will have the list of devices available. BIOS memory will have the next steps details like what is the first boot device it has to select etc. It will select the first boot device and gives back the control to Processor(CPU). Suppose if it does not find first boot device, it will check for next boot device, if not third and so on. If BIOS do not find any boot device it will alert user stating "No boot device found".

Stage 2: MBR loading

Once the BIOS gives control back to CPU, it will try to load MBR of the first boot device(We will consider it as HDD). MBR is a small part of Hard Disk with just a size of 512 Bytes, I repeat its just 512 Bytes. This MBR resides at the starting of HDD or end of HDD depending on manufacturer.

What is MBR?

MBR(Master Boot recorder) is a location on disk which have details about

- Primary boot loader code(This is of 446 Bytes)
- Partition table information(64 Bytes)
- Magic number(2 Bytes)

Which will be equal to 512B (446+64+2)B.

Primary Boot loader code: This code provides boot loader information and location details of actual boot loader code on the hard disk. This is helpful for CPU to load second stage of Boot loader.

Partition table: MBR contains 64 bytes of data which stores Partition table information such as what is the start and end of each partition, size of partition, type of partition(Whether it's a primary or extended etc). As we all know HDD support only 4 partitions, this is because of the limitation of its information in MBR. For a partition to represent in MBR, it requires 16 Bytes of space in it so at most we will get 4 partitions. Check our detail post on this concept to know more about this.

Magic Number: The magic number service as validation check for MBR. If MBR gets corrupted this magic number is used to retrieve it. What to take backup of your MBR try this.

Once your CPU knows all these details, it will try to analyse them and read the first portion of MBR to load Second stage of Boot loader

Stage 3: Boot loader Stage 2 (GRUB loader)

Once the Bootloader stage 1 is completed and able to find the actual bootloader location, Stage 1 bootloader start second stage by loading Bootloader into memory. In this stage GRUB(Grand Unified Bootloader) which is located in the first 30 kilobytes of hard disk immediately following the MBR is loaded into RAM for reading its configuration and displays the GRUB boot menu (where the user can manually specify the boot parameters) to the user. GRUB loads the user-selected (or default) kernel into memory and passes control on to the kernel. If user do not select the OS, after a defined timeout GRUB will load the default kernel in the memory for starting it.

Stage 4: Kernel

Once the control is given to kernel which is the central part of all your OS and act as a mediator of hardware and software components. Kernel once loaded into to RAM it always resides on RAM until the machine is shutdown. Once the Kernel starts its operations the first thing it do is executing INIT process.

Stage 5: INIT

This is the main stage of Booting Process

init(initialization) process is the root/parent process of all the process which run under Linux/Unix. The first process it runs is a script at /etc/rc.d/rc.sysinit which check all the system properties, hardware, display, SElinux, load kernel modules, file system check, file system mounting etc. Based on the appropriate run-level, scripts are executed to start/stop various processes to run the system and make it functional. INIT process read /etc/inittab which is an initialization table which defines starting of system programs. INIT will start each run level one after the other and start executing scripts corresponds to that runlevel. Know more about runlevels here. The script information is stored in different folders in /etc/ folder
/etc/rc0.d/ –Contain Start/Kill scripts which should be run in Runlevel 0
/etc/rc1.d/ –Contain Start/Kill scripts which should be run in Runlevel 1
/etc/rc2.d/ –Contain Start/Kill scripts which should be run in Runlevel 2
/etc/rc3.d/ –Contain Start/Kill scripts which should be run in Runlevel 3
/etc/rc4.d/ –Contain Start/Kill scripts which should be run in Runlevel 4
/etc/rc5.d/ –Contain Start/Kill scripts which should be run in Runlevel 5
/etc/rc6.d/ –Contain Start/Kill scripts which should be run in Runlevel 6

Know more about S and K convention used in the script names under /etc/rc*.d here.
Once the initialization process completes mandatory run level and reach to default runlevel set in /etc/inittab, init process run one more file /etc/rc.local which are the last commands run in initialization process or even booting process. Once everything is completed the control is given back to the kernel

Stage 6: User prompt

This is actually not part of booting process but thought of including it here for better understating. Once the Kernel get the control it start multiple instances of "getty" which waits for console logins which spawn one's user shell process and gives you user prompt to login.

LILO :

LILO (**LI**nux **LO**ader) is a boot loader for Linux and was the default boot loader for most Linux distributions in the years after the popularity of loadlin. Today, most distributions use GRUB as the default boot loader. LILO has been discontinued in December 2015, with a request for potential developers.

LILO does not depend on a specific file system, and can boot an operating system (e.g., Linux kernel images) from floppy disks and hard disks. One of up to sixteen different images can be selected at boot time. Various parameters, such as the root device, can be set independently for each kernel. LILO can be placed in the master boot record (MBR) or the boot sector of a partition. In the latter case, the MBR must contain code to load LILO.

The lilo.conf file is typically located at /etc/lilo.conf. Within lilo.conf there are typically two section types. The first section, which defines the global options, contains parameters which specify boot location attributes. The second section(s) contain parameters associated with the operating system images to be loaded.

When LILO loads itself it displays the word “LILO”. Each letter is printed before or after some specific action. If LILO fails at some point, the letters printed so far can be used to identify the problem.

(nothing)

No part of LILO has been loaded. LILO either isn't installed or the partition on which its boot sector is located isn't active. The boot media is incorrect or faulty.

L

The first stage boot loader has been loaded and started, but it can't load the second stage boot loader. The two-digit error codes indicate the type of problem. This condition usually indicates a media failure or bad disk parameters in the BIOS.

LI

The first stage boot loader was able to load the second stage boot loader, but has failed to execute it. This can be caused by bad disk parameters in the BIOS.

LIL

The second stage boot loader has been started, but it can't load the descriptor table from the map file. This is typically caused by a media failure or by bad disk parameters in the BIOS.

LIL?

The second stage boot loader has been loaded at an incorrect address. This is typically caused by bad disk parameters in the BIOS.

LIL-

The descriptor table is corrupt. This can be caused by bad disk parameters in the BIOS.

LILO

All parts of LILO have been successfully loaded.

GRUB :

GNU GRUB is a bootloader (can also be spelled boot loader) capable of loading a variety of free and proprietary operating systems. GRUB will work well with Linux, DOS, Windows, or BSD. GRUB stands for GRand Unified Bootloader.

GRUB is dynamically configurable. This means that the user can make changes during the boot time, which include altering existing boot entries, adding new, custom entries, selecting different kernels, or modifying initrd. GRUB also supports Logical Block Address mode. This means that if your computer has a fairly modern BIOS that can access more than 8GB (first 1024 cylinders) of hard disk space, GRUB will automatically be able to access all of it.

GRUB can be run from or be installed to any device (floppy disk, hard disk, CD-ROM, USB drive, network drive) and can load operating systems from just as many locations, including network drives.

What about LILO?

LILO supports only up to 16 different boot selections; GRUB supports an unlimited number of boot entries.

LILO cannot boot from network; GRUB can.

LILO must be written again every time you change the configuration file; GRUB does not.

LILO does not have an interactive command interface.

How does GRUB work?

When a computer boots, the BIOS transfers control to the first boot device, which can be a hard disk, a floppy disk, a CD-ROM, or any other BIOS-recognized device. We'll concentrate on hard disks, for the sake of simplicity.

The first sector on a hard is called the Master Boot Record (MBR). This sector is only 512 bytes long and contains a small piece of code (446 bytes) called the primary boot loader and the partition table (64 bytes) describing the primary and extended partitions.

By default, MBR code looks for the partition marked as active and once such a partition is found, it loads its boot sector into memory and passes control to it.

GRUB replaces the default MBR with its own code.

Furthermore, GRUB works in stages.

Stage 1 is located in the MBR and mainly points to Stage 2, since the MBR is too small to contain all of the needed data.

Stage 2 points to its configuration file, which contains all of the complex user interface and options we are normally familiar with when talking about GRUB. Stage 2 can be located anywhere on the disk. If Stage 2 cannot find its configuration table, GRUB will cease the boot sequence and present the user with a command line for manual configuration.

Stage 1.5 also exists and might be used if the boot information is small enough to fit in the area immediately after MBR.

The Stage architecture allows GRUB to be large (~20-30K) and therefore fairly complex and highly configurable, compared to most bootloaders, which are sparse and simple to fit within the limitations of the Partition Table.

Linux System Process Initialization (SysV)

Traditionally, Linux systems (and Unix systems before them) have used a sequential startup mechanism, which is described here. There is another page that describes a newer mechanism, known as Upstart and yet another page that describes an even newer mechanism known as systemd.

This older method of system initialization is still in wide use. Moreover, for compatibility reasons, much of its functionality is incorporated into upstart and systemd.

The method described on this page originated in a Unix version from AT&T Bell Labs, called "System V" where the "V" is Roman-numeral 5. "System V" is often abbreviated "SysV" and so that is what we use to distinguish this mechanism from the upstart and systemd mechanisms.

After the Linux kernel initialization is completed, the init script executes the program /sbin/init.

/sbin/init reads the file /etc/inittab which determines a series of scripts to be run to complete the initialization process:

1. The first script that init runs is /etc/rc.d/rc.sysinit. This script does several initialization tasks.
2. The next thing init does (again driven by entries in /etc/inittab) is run /etc/rc.d/rc, passing it a number known as the runlevel. Certain runlevels are standard:

   0 = halt
   1 = single-user mode
   6 = reboot

   Runlevels 2-5 are used for various forms of multi-user mode; the exact meaning of these runlevels differs among distros.

   The initial runlevel is specified in /etc/inittab.

   (The processing in this step can also be rerun by the system administrator after the boot process is finished in order to change the run level from the previously set level to a new run level. The administrator does this by running the command /sbin/init X (where X is the runlevel as above). Note that this program (/sbin/init) is the same program that runs during system initialization. If /sbin/init is run as process ID 1 then it does all of the processing described on this page. If run as any other process ID it simply sends a signal with the new runlevel to process ID 1 (the "real" init process) which runs the RC scripts as described in this step. In the case of the system administrator resetting the runlevel, the first step is to run the scripts in the directory /rc.d/rcX.d (where X is the *old* runlevel) whose name begin with "K". Each of these scripts is passed the parameter, "stop". These are actually symbolic links to files in /etc/rc.d/init.d.) This stops the programs that were running at the previous run level.

   Next the scripts in the directory /rc.d/rcX.d (where X is the *new* runlevel) whose name begin with "S" are run. Each of these scripts is passed the parameter, "start". These "S" files are also symbolic links to files in /etc/rc.d/init.d. This starts the programs that are to run at the new run level.

   Note: It would be wasteful or even harmful to stop and then restart a program that is needed on both the old and new runlevels. For that reason, this condition is detected and the stop and start steps are skipped programs that are on both runlevels.

Once the RC scripts are run, the init process then reads additional entries in /etc/inittab and uses them to start various local login mechanisms.

Init Run levels

SysV Init Runlevels

The SysV init runlevel system provides a standard process for controlling which programs `init` launches or halts when initializing a runlevel. SysV init was chosen because it is easier to use and more flexible than the traditional BSD-style init process.

The configuration files for SysV init are located in the `/etc/rc.d/` directory. Within this directory, are the `rc`, `rc.local`, `rc.sysinit`, and, optionally, the `rc.serial` scripts as well as the following directories:

```
init.d/ rc0.d/ rc1.d/ rc2.d/ rc3.d/ rc4.d/ rc5.d/ rc6.d/
```

The `init.d/` directory contains the scripts used by the `/sbin/init` command when controlling services. Each of the numbered directories represent the six runlevels configured by default under Red Hat Enterprise Linux.

Init Runlevels

The idea behind SysV init runlevels revolves around the idea that different systems can be used in different ways. For example, a server runs more efficiently without the drag on system resources created by the X Window System. Or there may be times when a system administrator may need to operate the system at a lower runlevel to perform diagnostic tasks, like fixing disk corruption in runlevel 1.

The characteristics of a given runlevel determine which services are halted and started by `init`. For instance, runlevel 1 (single user mode) halts any network services, while runlevel 3 starts these services. By assigning specific services to be halted or started on a given runlevel, `init` can quickly change the mode of the machine without the user manually stopping and starting services.

The following runlevels are defined by default under Red Hat Enterprise Linux:

- `0` — Halt
- `1` — Single-user text mode

  - **`2`** — Not used (user-definable)
  - **`3`** — Full multi-user text mode
  - **`4`** — Not used (user-definable)
  - **`5`** — Full multi-user graphical mode (with an X-based login screen)
  - **`6`** — Reboot

In general, users operate Red Hat Enterprise Linux at runlevel 3 or runlevel 5 — both full multi-user modes. Users sometimes customize runlevels 2 and 4 to meet specific needs, since they are not used.
The default runlevel for the system is listed in **`/etc/inittab`**. To find out the default runlevel for a system, look for the line similar to the following near the bottom of **`/etc/inittab`**:

```
id:5:initdefault:
```

The default runlevel listed in this example is five, as the number after the first colon indicates. To change it, edit **`/etc/inittab`** as root.

Warning

Be very careful when editing **`/etc/inittab`**. Simple typos can cause the system to become unbootable. If this happens, either use a boot CD or DVD, enter single-user mode, or enter rescue mode to boot the computer and repair the file.

It is possible to change the default runlevel at boot time by modifying the arguments passed by the boot loader to the kernel.

ext2

The **ext2** or **second extended filesystem** is a file system for the Linux kernel. It was initially designed by Rémy Card as a replacement for the extended file system (ext). Having been designed according to the same principles as the Berkeley Fast File System from BSD, it was the first commercial-grade filesystem for Linux.

Ext2 was the default filesystem in several Linux distributions, including Debian and Red Hat Linux, until supplanted more recently byext3, which is almost completely compatible with ext2 and is a journaling file system. ext2 is still the filesystem of choice for flash-based storage media (such as SD cards, and USB flash drives), since its lack of a journal increases performance and minimizes the number of writes, and flash devices have a limited number of write cycles. However, recent Linux kernels support a journal-less mode of ext4 which provides benefits not found with ext2.

ext2 data structures[edit]

The space in ext2 is split up into blocks. These blocks are grouped into block groups, analogous to cylinder groups in the Unix File System. There are typically thousands of blocks on a large file system. Data for any given file is typically contained within a single block group where possible. This is done to minimize the number of disk seeks when reading large amounts of contiguous data.

Each block group contains a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table and finally the actual data blocks.

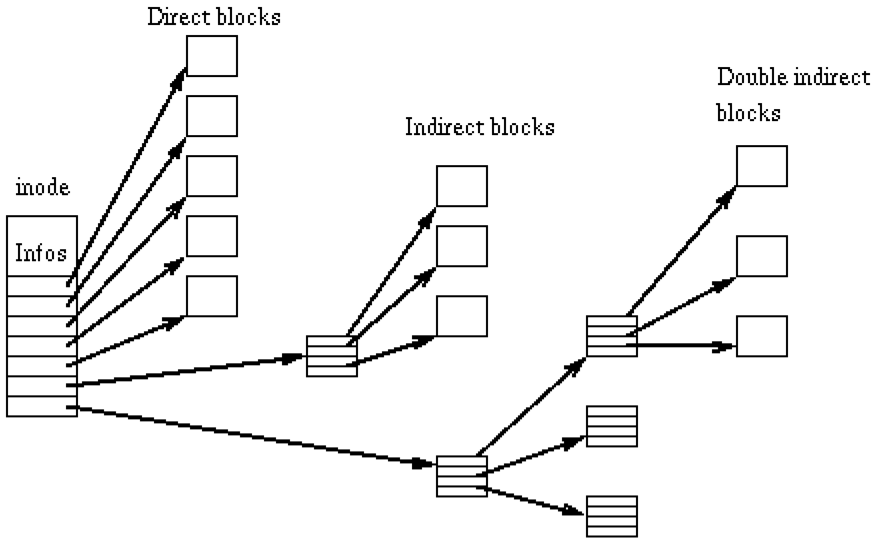
The superblock contains important information that is crucial to the booting of the operating system. Thus backup copies are made in multiple block groups in the file system. However, typically only the first copy of it, which is found at the first block of the file system, is used in the booting.

The group descriptor stores the location of the block bitmap, inode bitmap and the start of the inode table for every block group. These, in turn, are stored in a group descriptor table.

Inodes[edit]

Every file or directory is represented by an inode. The term "inode" comes from "index node" (over the time, it became i-node and then inode).[5] The inode includes data about the size, permission, ownership, and location on disk of the file or directory.

Example of ext2 inode structure:



Quote from the Linux kernel documentation for ext2:

"There are pointers to the first 12 blocks which contain the file's data in the inode. There is a pointer to an indirect block (which contains pointers to the next set of blocks), a pointer to a doubly indirect block and a pointer to a trebly indirect block."

So, there is a structure in ext2 that has 15 pointers. Pointers 1 to 12 point to direct blocks, pointer 13 points to an indirect block, pointer 14 points to a doubly indirect block, and pointer 15 points to a trebly indirect block.

Directories[edit]

Each directory is a list of directory entries. Each directory entry associates one file name with one inode number, and consists of the inode number, the length of the file name, and the actual text of the file name. To find a file, the directory is searched front-to-back for the associated filename. For reasonable directory sizes, this is fine. But for very large directories this is inefficient, and ext3 offers a second way of storing directories (HTree) that is more efficient than just a list of filenames.

The root directory is always stored in inode number two, so that the file system code can find it at mount time. Subdirectories are implemented by storing the name of the subdirectory in the name field, and the inode number of the subdirectory in the inode field. Hard links are implemented by storing the same inode number with more than one file name. Accessing the file by either name results in the same inode number, and therefore the same data.

ext3

ext3, or **third extended filesystem**, is a journaled file system that is commonly used by the Linux kernel. It is the default file systemfor many popular Linux distributions.

Advantages[edit]

The performance (speed) of ext3 is less attractive than competing Linux filesystems, such as ext4, JFS, ReiserFS and XFS. But ext3 has a significant advantage in that it allows in-place upgrades from ext2 without having to back up and restore data. Benchmarks suggest that ext3 also uses less CPU power than ReiserFS and XFS.[5][6] It is also considered safer than the other Linux file systems, due to its relative simplicity and wider testing base.[7][8]

ext3 adds the following features to ext2:

- A journal.
- Online file system growth.
- HTree indexing for larger directories.[9]

Without these features, any ext3 file system is also a valid ext2 file system. This situation has allowed well-tested and mature file system maintenance utilities for maintaining and repairing ext2 file systems to also be used with ext3 without major changes. The ext2 and ext3 file systems share the same standard set of utilities, e2fsprogs, which includes an fsck tool. The close relationship also makes conversion between the two file systems (both forward to ext3 and backward to ext2) straightforward.

ext3 lacks "modern" filesystem features, such as dynamic inode allocation and extents. This situation might sometimes be a disadvantage, but for recoverability, it is a significant advantage. The file system metadata is all in fixed, well-known locations, and data structures have some redundancy. In significant data corruption, ext2 or ext3 may be recoverable, while a tree-based file system may not.

Disadvantages[edit]

Functionality[edit]

Since ext3 aims to be backwards compatible with the earlier ext2, many of the on-disk structures are similar to those of ext2. Consequently, ext3 lacks recent features, such asextents, dynamic allocation of inodes, and block suballocation.[14] A directory can have at most 31998 subdirectories, because an inode can have at most 32000 links.[15]

ext3, like most current Linux filesystems, cannot be fsck-ed while the filesystem is mounted for writing. Attempting to check a file system that is already mounted may detect bogus errors where changed data has not reached the disk yet, and corrupt the file system in an attempt to "fix" these errors.

Defragmentation[edit]

There is no online ext3 defragmentation tool that works on the filesystem level. There is an offline ext2 defragmenter, `e2defrag`, but it requires that the ext3 filesystem be converted back to ext2 first. But `e2defrag` may destroy data, depending on the feature bits turned on in the filesystem; it does not know how to treat many of the newer ext3 features.[16]

There are userspace defragmentation tools, like Shake[17] and defrag.[18][19] Shake works by allocating space for the whole file as one operation, which will generally cause the allocator to find contiguous disk space. If there are files which are used at the same time, Shake will try to write them next to one another. Defrag works by copying each file over itself. However, this strategy works only if the file system has enough free space. A true defragmentation tool does not exist for ext3.[20]

However, as the Linux System Administrator Guide states, "Modern Linux filesystem(s) keep fragmentation at a minimum by keeping all blocks in a file close together, even if they can't be stored in consecutive sectors. Some filesystems, like ext3, effectively allocate the free block that is nearest to other blocks in a file. Therefore it is not necessary to worry about fragmentation in a Linux system."[21]

While ext3 is resistant to file fragmentation, ext3 can get fragmented over time or for specific usage patterns, like slowly writing large files.[22][23] Consequently, ext4 (the successor to ext3) has an online filesystem defragmentation utility e4defrag[24] and currently supports extents (contiguous file regions).

Undelete[edit]

ext3 does not support the recovery of deleted files. The ext3 driver actively deletes files by wiping file inodes,[25] for crash safety reasons.

There are still several techniques[26] and some free[27] and commercial[28] software for recovery of deleted or lost files using file system journal analysis; however, they do not guarantee any specific file recovery.

Compression[edit]

e3compr[29] is an unofficial patch for ext3 that does transparent compression. It is a direct port of e2compr and still needs further development. It compiles and boots well with upstream kernels[*citation needed*], but journaling is not implemented yet.

Lack of snapshots support[edit]

Unlike a number of modern file systems, ext3 does not have native support for snapshots—the ability to quickly capture the state of the filesystem at arbitrary times. Instead, it relies on less-space-efficient, volume-level snapshots provided by the Linux LVM. The Next3 file system is a modified version of ext3 which offers snapshots support, yet retains compatibility with the ext3 on-disk format.

Unix File System

The **Unix file system** (**UFS**; also called the **Berkeley Fast File System**, the **BSD Fast File System** or **FFS**) is a file system used by many Unix and Unix-like operating systems. It is a distant descendant of the original filesystem used by Version 7 Unix.

A UFS volume is composed of the following parts:

- A few blocks at the beginning of the partition reserved for boot blocks (which must be initialized separately from the filesystem)
- A superblock, containing a magic number identifying this as a UFS filesystem, and some other vital numbers describing this filesystem's geometry and statistics and behavioral tuning parameters
- A collection of cylinder groups. Each cylinder group has the following components:
  - A backup copy of the superblock
  - A cylinder group header, with statistics, free lists, etc., about this cylinder group, similar to those in the superblock
  - A number of inodes, each containing file attributes
  - A number of data blocks

Inodes are numbered sequentially, starting at 0. Inode 0 is reserved for unallocated directory entries, inode 1 was the inode of the bad block file in historical UNIX versions, followed by the inode for the root directory, which is always inode 2 and the inode for the lost+found directory which is inode 3.

Directory files contain only the list of filenames in the directory and the inode associated with each file. All file metadata is kept in the inode.

ReiserFS

ReiserFS is a general-purpose, journaled computer file system formerly designed and implemented by a team at Namesys led byHans Reiser. ReiserFS is currently supported on Linux (without quota support). Introduced in version 2.4.1 of the Linux kernel, it was the first journaling file system to be included in the standard kernel. ReiserFS is the default file system on the Elive, Xandros,Linspire, GoboLinux, and Yoper Linux distributions. ReiserFS was the default file system in Novell's SUSE Linux Enterprise until Novell decided to move to ext3 on October 12, 2006 for future releases.

Features[edit]

At the time of its introduction, ReiserFS offered features that had not been available in existing Linux file systems:

- Metadata-only journaling (also block journaling, since Linux 2.6.8), its most-publicized advantage over what was the stock Linux file system at the time, ext2.
- Online resizing (growth only), with or without an underlying volume manager such as LVM. Since then, Namesys has also provided tools to resize (both grow and shrink) ReiserFS file systems offline.
- Tail packing, a scheme to reduce internal fragmentation. Tail packing, however, can have a significant performance impact. Reiser4 may have improved this by packing tails where it does not hurt performance.[5]

JFS (file system)

Journaled File System or **JFS** is a 64-bit journaling file system created by IBM. There are versions for AIX, eComStation, OS/2, andLinux operating systems. The latter is available as free software under the terms of the GNU General Public License (GPL). HP-UXhas another, different filesystem named JFS that is actually an OEM version of Veritas Software's VxFS.

Features[edit]

JFS supports the following features.[8][9]

Journal[edit]

JFS is a journaling file system. Rather than adding journaling as an add-on feature like in the ext3 file system, it was implemented from the start. The journal can be up to 128MB. JFS journals metadata only, which means that metadata will remain consistent but user files may be corrupted after a crash or power loss. JFS' journaling is similar to XFS where it only journals parts of the inode.[10]

B+ Tree[edit]

JFS uses a B+ tree to accelerate lookups in directories. JFS can store 8 entries of a directory in the directory's inode before moving the entries to a B+ tree. JFS also indexes extents in a B+ tree.

Dynamic Inode Allocation[edit]

JFS dynamically allocates space for disk inodes as necessary. Each inode is 512 Bytes. 32 Inodes are allocated on a 16KB Extent.

Extents[edit]

JFS allocates files as an extent. An extent is a variable-length sequence of Aggregate blocks. An extent may be located in several allocation groups. To solve this the extents are indexed in a B+ tree for better performance when locating the extent locations.

Compression[edit]

Compression is supported only in JFS1 on AIX and uses a variation of the LZ algorithm. Because of high CPU usage and increased free space fragmentation, compression is not recommended for use other than on a single user workstation or off-line backup areas.[2][11]

Concurrent Input / Output (CIO)[edit]

JFS normally applies read-shared, write-exclusive locking to files, which avoids data inconsistencies but imposes write serialization at the file level. The CIO option disables this locking. Applications such as relational databases which maintain data consistency themselves can use this option to largely eliminate filesystem overheads.[12]

Allocation Groups[edit]

JFS uses Allocation groups. Allocation groups divide the aggregate space into chunks. This allows JFS to use resource allocation policies to achieve great I/O performance. The first policy is to try to cluster disk blocks and disk inodes for related data in the same AG in order to achieve good locality for the disk. The second policy is to distribute unrelated data throughout the file system in an attempt to minimize free-space fragmentation. When there is an open file JFS will lock the AG the file resides in and only allow the open file to grow. This reduces fragmentation as only the open file can write to the AG.

JFS Superblocks[edit]

The superblock maintains information about the entire file system and includes the following fields:

- Size of the file system
- Number of data blocks in the file system
- A flag indicating the state of the file system
- Allocation group sizes
- File system block size

JFS in Linux[edit]

In the Linux operating system, JFS is supported with the kernel module (since the kernel version *2.4.18pre9-ac4*) and the complementary userspace utilities packaged under the name *JFSutils*. Most Linux distributions support JFS, unless it is specifically removed due to space restrictions or other concerns. Most LiveCD distributions do not provide support of JFS because JFSutils are not installed.

According to reviews and benchmarks of the available filesystems for Linux, JFS is fast and reliable, with consistently good performance under different kinds of load, contrary to other filesystems that seem to perform better under particular usage patterns, for instance with small or large files. Another characteristic often mentioned, is that it's light and efficient with available system resources and even heavy disk activity is realized with low CPU usage. Especially for databases which need synchronous writes to survive a hardware crash, JFS with external journal seems to be the best option. [13][14][15] File fragmentation on JFS impairs filesystem performance less than on more traditional Linux ext3 filesystems.[16]

Actual usage of JFS in Linux is uncommon, as ext4 typically offers better performance.[17] There are also potential problems with JFS, such as its implementation of journal writes. They can be postponed until there is another trigger - potentially indefinitely, which can cause data loss over a theoretically infinite timeframe.[18]

The Virtual File System (VFS)

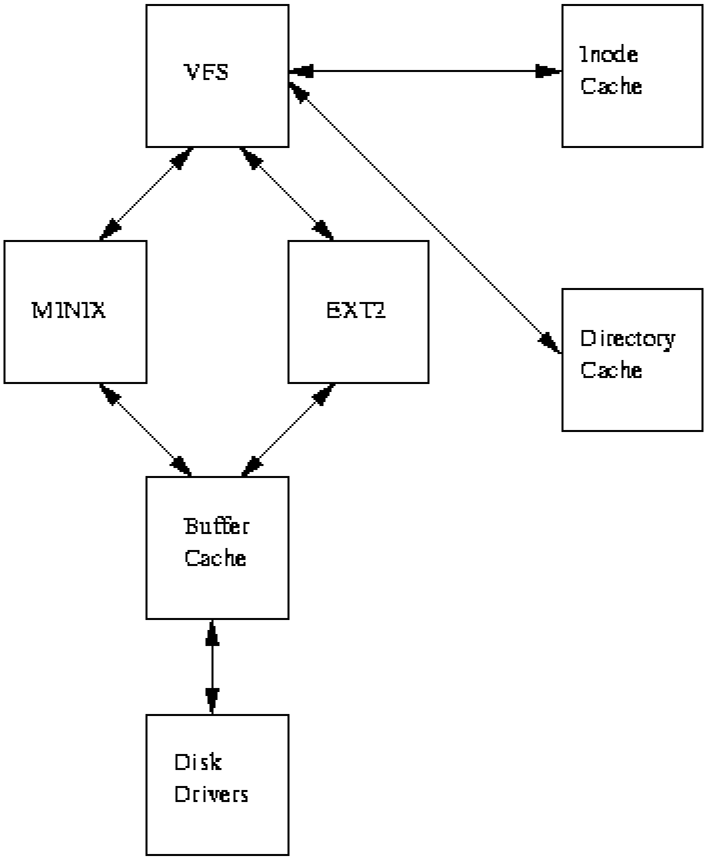


Figure 9.4: A Logical Diagram of the Virtual File System

Figure 9.4 shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the `VFAT` file system is implemented as a kernel module, then it is only loaded when a `VFAT` file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions. So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ``/'' directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

As the system's processes access directories and files, system routines are called that traverse the VFS inodes in the system.

For example, typing ls for a directory or cat for a file cause the the Virtual File System to search through the VFS inodes that represent the file system. As every file and directory on the system is represented by a VFS inode, then a number of inodes will be being repeatedly accessed. These inodes are kept in the inode cache which makes access to them quicker. If an inode is not in the inode cache, then a file system specific routine must be called in order to read the appropriate inode. The action of reading the inode causes it to be put into the inode cache and further accesses to the inode keep it in the cache. The less used VFS inodes get removed from the cache.

All of the Linux file systems use a common buffer cache to cache data buffers from the underlying devices to help speed up access by all of the file systems to the physical devices holding the file systems.

This buffer cache is independent of the file systems and is integrated into the mechanisms that the Linux kernel uses to allocate and read and write data buffers. It has the distinct advantage of making the Linux file systems independent from the underlying media and from the device drivers that support them. All block structured devices register themselves with the Linux kernel and present a uniform, block based, usually asynchronous interface. Even relatively complex block devices such as SCSI devices do this. As the real file systems read data from the underlying physical disks, this results in requests to the block device drivers to read physical blocks from the device that they control. Integrated into this block device interface is the buffer cache. As blocks are read by the file systems they are saved in the global buffer cache shared by all of the file systems and the Linux kernel. Buffers within it are identified by their block number and a unique identifier for the device that read it. So, if the same data is needed often, it will be retrieved from the buffer cache rather than read from the disk, which would take somewhat longer. Some devices support read ahead where data blocks are speculatively read just in case they are needed.

The VFS also keeps a cache of directory lookups so that the inodes for frequently used directories can be quickly found.

As an experiment, try listing a directory that you have not listed recently. The first time you list it, you may notice a slight pause but the second time you list its contents the result is immediate. The directory cache does not store the inodes for the directories itself; these should be in the inode cache, the directory cache simply stores the mapping between the full directory names and their inode numbers.