Press the power button on your system, and after few moments you see the Linux login prompt.

Have you ever wondered what happens behind the scenes from the time you press the power button until the Linux login prompt appears?

The following are the 6 high level stages of a typical Linux boot process.  
  


**1. BIOS**

* BIOS stands for Basic Input/Output System
* Performs some system integrity checks
* Searches, loads, and executes the boot loader program.
* It looks for boot loader in floppy, cd-rom, or hard drive. You can press a key (typically F12 of F2, but it depends on your system) during the BIOS startup to change the boot sequence.
* Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
* So, in simple terms BIOS loads and executes the MBR boot loader.

**2. MBR**

* MBR stands for Master Boot Record.
* It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
* MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
* It contains information about GRUB (or LILO in old systems).
* So, in simple terms MBR loads and executes the GRUB boot loader.

**3. GRUB**

* GRUB stands for Grand Unified Bootloader.
* If you have multiple kernel images installed on your system, you can choose which one to be executed.
* GRUB displays a splash screen, waits for few seconds, if you don’t enter anything, it loads the default kernel image as specified in the grub configuration file.
* GRUB has the knowledge of the filesystem (the older Linux loader LILO didn’t understand filesystem).
* Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

#boot=/dev/sda

default=0

timeout=5

splashimage=(hd0,0)/boot/grub/splash.xpm.gz

hiddenmenu

title CentOS (2.6.18-194.el5PAE)

root (hd0,0)

kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/

initrd /boot/initrd-2.6.18-194.el5PAE.img

* As you notice from the above info, it contains kernel and initrd image.
* So, in simple terms GRUB just loads and executes Kernel and initrd images.

**4. Kernel**

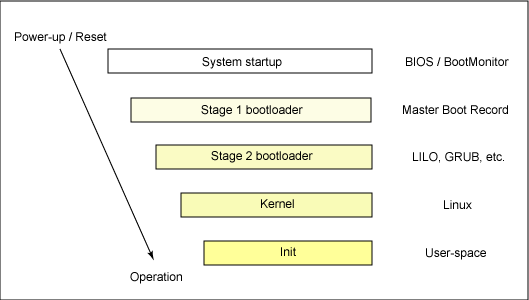
* Mounts the root file system as specified in the “root=” in grub.conf
* Kernel executes the /sbin/init program
* Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a ‘ps -ef | grep init’ and check the pid.
* initrd stands for Initial RAM Disk.
* initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

**5. Init**

* Looks at the /etc/inittab file to decide the Linux run level.
* Following are the available run levels
  + 0 – halt
  + 1 – Single user mode
  + 2 – Multiuser, without NFS
  + 3 – Full multiuser mode
  + 4 – unused
  + 5 – X11
  + 6 – reboot
* Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
* Execute ‘grep initdefault /etc/inittab’ on your system to identify the default run level
* If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
* Typically you would set the default run level to either 3 or 5.

**6. Runlevel programs**

* When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
* Depending on your default init level setting, the system will execute the programs from one of the following directories.
  + Run level 0 – /etc/rc.d/rc0.d/
  + Run level 1 – /etc/rc.d/rc1.d/
  + Run level 2 – /etc/rc.d/rc2.d/
  + Run level 3 – /etc/rc.d/rc3.d/
  + Run level 4 – /etc/rc.d/rc4.d/
  + Run level 5 – /etc/rc.d/rc5.d/
  + Run level 6 – /etc/rc.d/rc6.d/
* Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
* Under the /etc/rc.d/rc\*.d/ directories, you would see programs that start with S and K.
* Programs starts with S are used during startup. S for startup.
* Programs starts with K are used during shutdown. K for kill.
* There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
* For example, S12syslog is to start the syslog deamon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

**Figure 1. The 20,000-foot view of the Linux boot process**  


When a system is first booted, or is reset, the processor executes code at a well-known location. In a personal computer (PC), this location is in the basic input/output system (BIOS), which is stored in flash memory on the motherboard. The central processing unit (CPU) in an embedded system invokes the reset vector to start a program at a known address in flash/ROM. In either case, the result is the same. Because PCs offer so much flexibility, the BIOS must determine which devices are candidates for boot. We'll look at this in more detail later.

When a boot device is found, the first-stage boot loader is loaded into RAM and executed. This boot loader is less than 512 bytes in length (a single sector), and its job is to load the second-stage boot loader.

When the second-stage boot loader is in RAM and executing, a splash screen is commonly displayed, and Linux and an optional initial RAM disk (temporary root file system) are loaded into memory. When the images are loaded, the second-stage boot loader passes control to the kernel image and the kernel is decompressed and initialized. At this stage, the second-stage boot loader checks the system hardware, enumerates the attached hardware devices, mounts the root device, and then loads the necessary kernel modules. When complete, the first user-space program (init) starts, and high-level system initialization is performed.

That's Linux boot in a nutshell. Now let's dig in a little further and explore some of the details of the Linux boot process.

The system startup stage depends on the hardware that Linux is being booted on. On an embedded platform, a bootstrap environment is used when the system is powered on, or reset. Examples include U-Boot, RedBoot, and MicroMonitor from Lucent. Embedded platforms are commonly shipped with a boot monitor. These programs reside in special region of flash memory on the target hardware and provide the means to download a Linux kernel image into flash memory and subsequently execute it. In addition to having the ability to store and boot a Linux image, these boot monitors perform some level of system test and hardware initialization. In an embedded target, these boot monitors commonly cover both the first- and second-stage boot loaders.

## Extracting the MBR

To see the contents of your MBR, use this command:

# **dd if=/dev/hda of=mbr.bin bs=512 count=1**   
# **od -xa mbr.bin**

The dd command, which needs to be run from root, reads the first 512 bytes from /dev/hda (the first Integrated Drive Electronics, or IDE drive) and writes them to the mbr.bin file. The od command prints the binary file in hex and ASCII formats.

In a PC, booting Linux begins in the BIOS at address 0xFFFF0. The first step of the BIOS is the power-on self test (POST). The job of the POST is to perform a check of the hardware. The second step of the BIOS is local device enumeration and initialization.

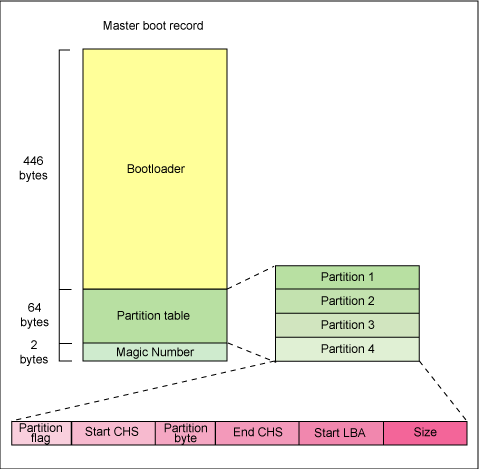
Given the different uses of BIOS functions, the BIOS is made up of two parts: the POST code and runtime services. After the POST is complete, it is flushed from memory, but the BIOS runtime services remain and are available to the target operating system.

To boot an operating system, the BIOS runtime searches for devices that are both active and bootable in the order of preference defined by the complementary metal oxide semiconductor (CMOS) settings. A boot device can be a floppy disk, a CD-ROM, a partition on a hard disk, a device on the network, or even a USB flash memory stick.

Commonly, Linux is booted from a hard disk, where the Master Boot Record (MBR) contains the primary boot loader. The MBR is a 512-byte sector, located in the first sector on the disk (sector 1 of cylinder 0, head 0). After the MBR is loaded into RAM, the BIOS yields control to it.

Stage 1 boot loader

The primary boot loader that resides in the MBR is a 512-byte image containing both program code and a small partition table (see Figure 2). The first 446 bytes are the primary boot loader, which contains both executable code and error message text. The next sixty-four bytes are the partition table, which contains a record for each of four partitions (sixteen bytes each). The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR.

**Figure 2. Anatomy of the MBR**  


The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot record is read from the device into RAM and executed.

Stage 2 boot loader

The secondary, or second-stage, boot loader could be more aptly called the kernel loader. The task at this stage is to load the Linux kernel and optional initial RAM disk.

## GRUB stage boot loaders

The /boot/grub directory contains the stage1, stage1.5, and stage2 boot loaders, as well as a number of alternate loaders (for example, CR-ROMs use the iso9660\_stage\_1\_5).

The first- and second-stage boot loaders combined are called Linux Loader (LILO) or GRand Unified Bootloader (GRUB) in the x86 PC environment. Because LILO has some disadvantages that were corrected in GRUB, let's look into GRUB. (See many additional resources on GRUB, LILO, and related topics in the [Resources](http://www.ibm.com/developerworks/library/l-linuxboot/#resources) section later in this article.)

The great thing about GRUB is that it includes knowledge of Linux file systems. Instead of using raw sectors on the disk, as LILO does, GRUB can load a Linux kernel from an ext2 or ext3 file system. It does this by making the two-stage boot loader into a three-stage boot loader. Stage 1 (MBR) boots a stage 1.5 boot loader that understands the particular file system containing the Linux kernel image. Examples include reiserfs\_stage1\_5 (to load from a Reiser journaling file system) or e2fs\_stage1\_5 (to load from an ext2 or ext3 file system). When the stage 1.5 boot loader is loaded and running, the stage 2 boot loader can be loaded.

With stage 2 loaded, GRUB can, upon request, display a list of available kernels (defined in /etc/grub.conf, with soft links from /etc/grub/menu.lst and /etc/grub.conf). You can select a kernel and even amend it with additional kernel parameters. Optionally, you can use a command-line shell for greater manual control over the boot process.

With the second-stage boot loader in memory, the file system is consulted, and the default kernel image and initrd image are loaded into memory. With the images ready, the stage 2 boot loader invokes the kernel image.

Kernel

## Manual boot in GRUB

From the GRUB command-line, you can boot a specific kernel with a named initrd image as follows:

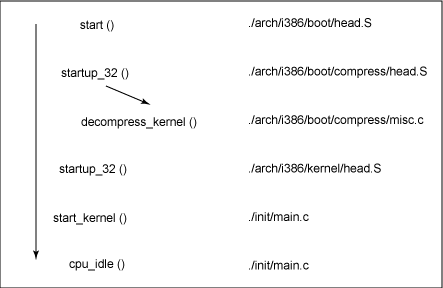
grub> **kernel /bzImage-2.6.14.2**  
  [Linux-bzImage, setup=0x1400, size=0x29672e]  
  
grub> **initrd /initrd-2.6.14.2.img**  
  [Linux-initrd @ 0x5f13000, 0xcc199 bytes]  
  
grub> **boot**  
  
Uncompressing Linux... Ok, booting the kernel.

If you don't know the name of the kernel to boot, just type a forward slash (/) and press the Tab key. GRUB will display the list of kernels and initrd images.

With the kernel image in memory and control given from the stage 2 boot loader, the kernel stage begins. The kernel image isn't so much an executable kernel, but a compressed kernel image. Typically this is a zImage (compressed image, less than 512KB) or a bzImage (big compressed image, greater than 512KB), that has been previously compressed with zlib. At the head of this kernel image is a routine that does some minimal amount of hardware setup and then decompresses the kernel contained within the kernel image and places it into high memory. If an initial RAM disk image is present, this routine moves it into memory and notes it for later use. The routine then calls the kernel and the kernel boot begins.

When the bzImage (for an i386 image) is invoked, you begin at ./arch/i386/boot/head.S in the start assembly routine (see Figure 3 for the major flow). This routine does some basic hardware setup and invokes the startup\_32 routine in ./arch/i386/boot/compressed/head.S. This routine sets up a basic environment (stack, etc.) and clears the Block Started by Symbol (BSS). The kernel is then decompressed through a call to a C function called decompress\_kernel (located in ./arch/i386/boot/compressed/misc.c). When the kernel is decompressed into memory, it is called. This is yet another startup\_32 function, but this function is in ./arch/i386/kernel/head.S.

In the new startup\_32 function (also called the swapper or process 0), the page tables are initialized and memory paging is enabled. The type of CPU is detected along with any optional floating-point unit (FPU) and stored away for later use. The start\_kernel function is then invoked (init/main.c), which takes you to the non-architecture specific Linux kernel. This is, in essence, the main function for the Linux kernel.

**Figure 3. Major functions flow for the Linux kernel i386 boot**   


With the call to start\_kernel, a long list of initialization functions are called to set up interrupts, perform further memory configuration, and load the initial RAM disk. In the end, a call is made to kernel\_thread (in arch/i386/kernel/process.c) to start the init function, which is the first user-space process. Finally, the idle task is started and the scheduler can now take control (after the call to cpu\_idle). With interrupts enabled, the pre-emptive scheduler periodically takes control to provide multitasking.

During the boot of the kernel, the initial-RAM disk (initrd) that was loaded into memory by the stage 2 boot loader is copied into RAM and mounted. This initrd serves as a temporary root file system in RAM and allows the kernel to fully boot without having to mount any physical disks. Since the necessary modules needed to interface with peripherals can be part of the initrd, the kernel can be very small, but still support a large number of possible hardware configurations. After the kernel is booted, the root file system is pivoted (via pivot\_root) where the initrd root file system is unmounted and the real root file system is mounted.

## decompress\_kernel output

The decompress\_kernel function is where you see the usual decompression messages emitted to the display:

Uncompressing Linux... Ok, booting the kernel.

The initrd function allows you to create a small Linux kernel with drivers compiled as loadable modules. These loadable modules give the kernel the means to access disks and the file systems on those disks, as well as drivers for other hardware assets. Because the root file system is a *file system* on a disk, the initrd function provides a means of bootstrapping to gain access to the disk and mount the real root file system. In an embedded target without a hard disk, the initrd can be the final root file system, or the final root file system can be mounted via the Network File System (NFS).

Init

After the kernel is booted and initialized, the kernel starts the first user-space application. This is the first program invoked that is compiled with the standard C library. Prior to this point in the process, no standard C applications have been executed.

In a desktop Linux system, the first application started is commonly /sbin/init. But it need not be. Rarely do embedded systems require the extensive initialization provided by init (as configured through /etc/inittab). In many cases, you can invoke a simple shell script that starts the necessary embedded applications.

Much like Linux itself, the Linux boot process is highly flexible, supporting a huge number of processors and hardware platforms. In the beginning, the loadlin boot loader provided a simple way to boot Linux without any frills. The LILO boot loader expanded the boot capabilities, but lacked any file system awareness. The latest generation of boot loaders, such as GRUB, permits Linux to boot from a range of file systems (from Minix to Reiser).