The Linux Boot Process

Written by Daniel Eriksen as a presentation for
the Bruce Grey Linux Users Group
[http://www.bglug.ca/]
April 06, 2004

Intro

Some points I make will be fairly technical in nature, but they will usually explain why certain things happen the way they do.

I will be covering the boot process in five stages. The BIOS, boot sectors, kernel, the init process and lastly, the SysVinit scripts. Of particular interest are the scripts that are run at the end. This is where one can make changes and customize the system to their needs.

Note that parts of this discussion will be specific to the x86 architecture.

BIOS

The computer begins the boot sequence by resetting the CPU. This sets several registers to fixed values and then executes the code found at a specific address. This address is mapped in hardware to a ROM chip that contains the BIOS. This ROM chip is usually an EEPROM, which allows it to be electrically erased and reprogrammed, like when you upgrade or "flash", your BIOS.

ROM - Read Only Memory

BIOS - Basic Input Output System

EEPROM - Electrically Erasable Programmable ROM. Also known as Flash ROM.

CMOS - Complimentary Metal Oxide Semiconductor

Your BIOS will usually provide you with a way to tell it about the different hardware that is attached. These settings are stored in the CMOS which uses a battery to maintain it's contents. A CMOS battery will usually outlive the motherboard it's attached to, but battery replacements are sometimes necessary.

The BIOS provides a number of routines and services that are mostly useless to the Linux kernel. A large part of the BIOS exists to support legacy systems like DOS. There is an obvious advantage here to eliminating this excess code and the LinuxBIOS project shows just how much advantage there is.

LinuxBIOS [http://www.linuxbios.org/] is an effort to provide the bare minimum to initialize the motherboard and allow a Linux kernel to boot. On some boards, stripping out these excess functions has made room for the Linux kernel, allowing it to be run directly from EEPROM. One humourous bug that had to be worked around is that some hard drives are unable to initialize themselves fast enough for the kernel.

There are basically four things the BIOS does at this point.

- The Power On Self Test, or POST routine runs to find certain hardware and to test that the hardware is working at a basic level.
- Hardware devices are initialized. PCI devices need to be given IRQs, input/output addresses and a table of these devices is then displayed on-screen.
- The BIOS must search for an operating system on the various media that is supported. This usually occurs by searching for a boot sector on the devices that are specified in the CMOS.
- Once a valid boot sector has been found, it is copied into RAM and then executed.

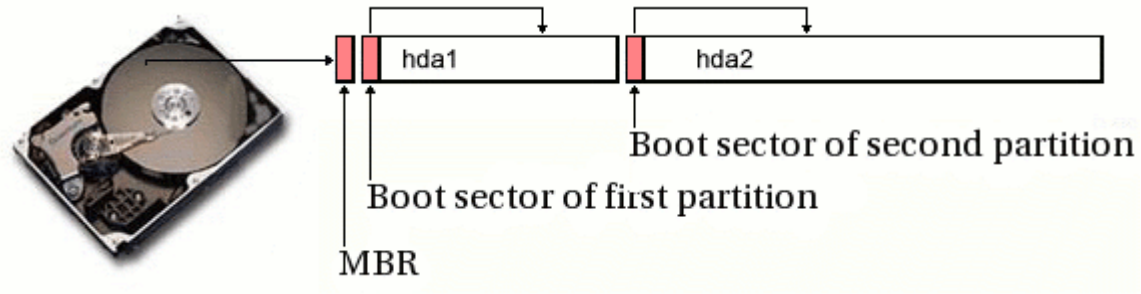
Boot Sectors

A sector has a length of 512 bytes. A sector becomes a "boot" sector because of it's location and the hex value 0xaa55 in the final two bytes. The BIOS looks for this value when scanning potential boot media.

When one creates a bootable floppy, the kernel created has a sector tacked onto the beginning of it. When this kernel is written to the beginning of a floppy, this first sector becomes the boot sector. It is this boot sector that displays the "Loading" message, and then proceeds to load the kernel.

MBR - Main Boot Record

A hard disk can have a boot sector in the first sector of the drive (which is known as the Main Boot Record) or in the first sector of a primary partition. It is also possible for extended partitions to hold a boot sector, but this must be supported by the boot manager.

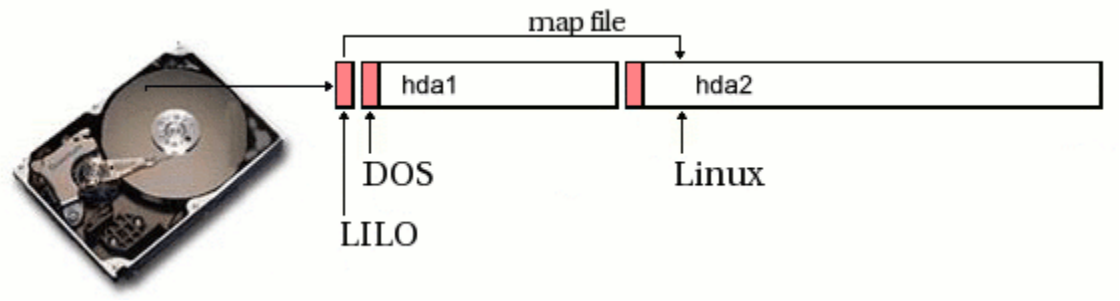


By default, a hard drive will usually contain the DOS Main Boot Record. The Linux Loader (also known as LILO) can be used instead of this default. Some people like to leave the Main Boot Record as is and just install LILO on a partition that is set as active. LILO supports booting many kinds of operating systems so there is an advantage to making it your Main Boot Record.

From the point of view of the BIOS, there is no such thing as a bootable partition. It is the Main Boot Record that allows a partition to be bootable. The Main Boot Record simply chain-loads the code found in a bootable partition. If the Main Boot Record contains the default DOS-MBR, then it tries to boot the partition marked 'active'. This active partition can be set using the Linux or DOS fdisk program. LILO does not

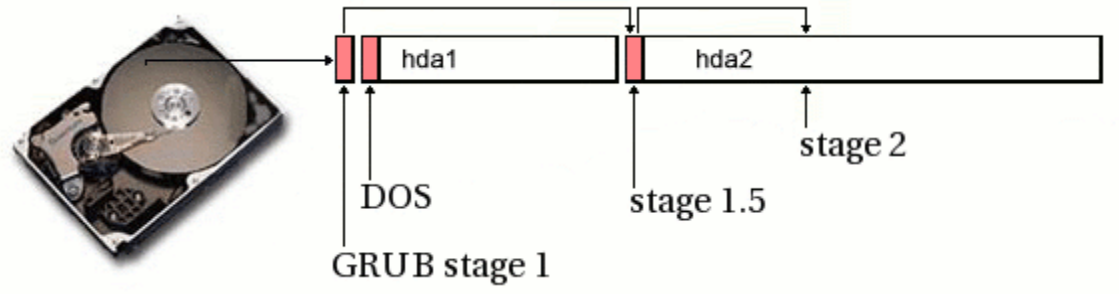
understand 'active' partitions, but can also chain-load a bootable partition. LILO is hard-coded with it's boot entries.

LILO is too large to fit into a single sector, so it is loaded in stages. It's progress can be tracked by the letters it prints on the screen. The first stage loader is what is found in the boot sector. It simply prints out an "L" and then loads the second stage loader. Once the second stage loader has been loaded, LILO prints an "I" and then executes the second stage. The second stage loader prints an "L" and then attempts to read a map file from disk. This map file tells LILO what operating systems are available. The map file is created when the 'lilo' command is executed at the command prompt or when one installs Linux. If the map file loads properly and is not corrupt, LILO prints out an "O", showing that it has successfully loaded.



When LILO is told to boot a Linux kernel it reads it directly from the disk. It knows the kernels location because of information found in the map file. Because LILO requires this map file to point to a hard-coded location on the hard drive, as soon as one moves the kernel or the disk geometry changes, LILO needs to be run to generate a new map file. A new boot loader has been designed to overcome this problem.

The GRand Unified Bootloader, otherwise known as 'GRUB', was written to address the need for a standard method of booting an operating system. This standard is known as the "Multiboot Specification" and GRUB is capable of booting a compatible kernel. GRUB is also able to boot a myriad of other operating systems including DOS, Windows, FreeBSD and Linux (which, incidentally, is not currently multiboot compatible).



Like LILO, GRUB also boots in stages due to it's size. Stage 1 is what is present in the main boot record and merely loads the next stage. Stage 1.5 is also quite

small, but is able to understand filesystems. Depending what filesystem contains the Stage 2 loader, a different Stage 1.5 is used, and is even optional in some circumstances.

Stage 2 is the meat of the loader and contains many features and options that can be explored. Because GRUB understands partitions and filesystems, it can load a kernel that it wasn't told about previously. No map file is necessary and GRUB does not need to be re-run after installing a new kernel.

Kernel

Many Linux distributions are shipping default kernels that support as much hardware as possible. In the past this has meant that the kernel image would be huge, containing support for as many bootable devices as possible. They were unable to make the kernel completely modular because sometimes these modules would be needed to mount the filesystem that contained the modules, a real chicken-egg problem.

The solution was to have the kernel load an initial RAM disk image, also known as an 'initrd' image, that contained many of the modules that would be needed to boot. Most of these modules will be SCSI controller drivers and support for various filesystems.

At this point, the BIOS has selected the boot device and it's boot sector has been loaded. The boot manager nows loads the kernel image and possibly an initial RAM disk image. Once loaded into RAM, the kernel is executed and the setup code runs.

The kernel must initialize any devices the system has. Even devices that have been initialized by the BIOS must be reinitialized. This provides portability and robustness by ensuring that each system has been initialized in a similar fashion, independent of the BIOS.

The next step that is performed by the setup code is switching the CPU from Real Mode to Protected Mode. This is where the CPU stops behaving like an ancient XT, which can only access one mebibyte of address space. Once in Protected Mode, privilege levels are assigned to running processes. This allows the operating system to be protected from normal user programs.

The setup code now loads the compressed kernel and calls the `decompress_kernel()` function. It is at this point that you will see the "Uncompressing Linux..." message printed on the screen.

The decompressed kernel now takes over and begins to set up the execution environment for the first Linux process. The kernel will now begin printing a large number of messages on the screen as it initializes the scheduler, irqs, console,

hardware, etc. This is done in the `start_kernel()` function and nearly every kernel component is initialized by this function.

The `kernel_thread()` function is called next to start init. The kernel goes into an idle loop and becomes an idle thread with process ID 0.

```
                  Simplified view of kernel execution
setup
    \---> switch to Protected Mode
           \---> decompress_kernel()
                   \---> start_kernel()            <= PID 0
                          \---> kernel_thread()
                                  \---> init        <= PID 1
```

Init

The first program that is run under the kernel is init. This program is always process 1. The Linux kernel can be told which program to use as init by passing the "`init=`" boot parameter. If this parameter is not specified, then the kernel will try to execute '`/sbin/init`', '`/etc/init`', '`/bin/init`' or '`/bin/sh`' in that order. If none of these exist, then the kernel will panic.

There are alternatives to using the init included with your system. You can actually place any executable in place of init. If one was building an embedded system, they could replace init with a program written in C that would run faster and could be streamlined for the system. The script based startup used by most systems is much easier to use, though, because of how easy it is to make changes.

Most systems use System V init. This system starts by running the init program. The init program reads it's configuration from the '`inittab`' file that is located in the 'etc' directory. This file is where runlevels come into existence. By editing this file, one could make their system use only three runlevels or could add some if they wished.

Runlevels have not been standardized in any way, but most Linux distributions use a similar layout.

Example inittab file

```
id:3:initdefault:                   <-- the default runlevel is defined

si::sysinit:/etc/init.d/rcS         <-- some tasks that are always run
l0:0:wait:/etc/init.d/rc 0          <-- runlevels 0 to 6 and their
l1:S1:wait:/etc/init.d/rc 1              associated scripts
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6

ca:12345:ctrlaltdel:/sbin/shutdown -h now  <-- trap CTRL-ALT-DEL
su:S016:respawn:/sbin/sulogin      <-- run sulogin on runlevels S, 0, 1, 6

1:2345:respawn:/sbin/agetty tty1 9600      <-- terminal 1
2:2345:respawn:/sbin/agetty tty2 9600
3:2345:respawn:/sbin/agetty tty3 9600
4:2345:respawn:/sbin/agetty tty4 9600
#5:2345:respawn:/sbin/agetty tty5 9600     <-- Terminals 5 and 6
#6:2345:respawn:/sbin/agetty tty6 9600          have been disabled
```

The main settings that are necessary are the specifying of the default runlevel, any terminals (usually six terminals are defined) and the scripts that are needed to be run for each runlevel. There is also a line specifying a script to run before all the others, like setting up localnet and mounting filesystems.

Many of the entries are also specified with '`respawn`'. This parameter tells init to continually execute this program whenever the previous one exits. When one logs out, init respawns the getty process to allow another login.

'`inittab`' is also where `CTRL-ALT-DEL` can be trapped to instead run the shutdown command. Some systems also run XFree86 from this file while systems like Debian, run XFree86 from the runlevel-specific scripts.

Runlevel Scripts

Linux generally has eight runlevels. Runlevel 0,1,6 and S are reserved for specific functions. More runlevels can be defined, but Unix has traditionally only used up to runlevel 6. Changing to runlevel 0 will halt the system and changing to runlevel 6 will cause a reboot. Runlevel S is for single user mode. It's important to note that runlevel S is not the same as runlevel 1. Runlevel 1 will run some scripts and then enter runlevel S. No scripts are run when entering runlevel S. It is for this reason that runlevel S is not meant to be entered directly with an already running system. Single user mode is most useful for making repairs or system changes.

The runlevel scripts can be setup in many different ways. I will be describing the layout used by Debian and many other systems. This layout is not much different then the others, but differences exist. I would encourage anyone who is interested, to take a look at their own system and trace out the process of booting these scripts. Having a good understanding of the boot scripts used on your system will aid greatly in your ability to understand the system as a whole.

At this point init is running and a default runlevel has been specified. The initial sysinit script has been run which has done things like setup networking, set the time, check filesystems and then mount them. A swap file will be mounted at this point as well. init now calls the script for the default runlevel. On my Debian system, the default runlevel is 2 which results in the 'rc' script being run with 2 as a parameter. This script and pretty much all the other boot scripts, are located in '`/etc/init.d`'. Each runlevel is given a directory to hold scripts for that runlevel and these directories are labeled rc0.d, rc1.d, rc2.d up to rc6.d. But having a copy of every script in each runlevel would be an administrative nightmare for maintaining, so these directories contain symbolic links that point to the real scripts. The real scripts are located in the '`/etc/init.d`' directory.

** Note that Red Hat and Mandrake place their files and directories in '`/etc/rc.d`' instead of '`/etc`'.

The symbolic links that are contained in these directories follow a specific naming convention.

`<action><2 digit number><original name>`
`example:` **`S40httpd`**

The link names will begin with an `<action>` which will either be an 'S' or a 'K' that will signify whether to 'start' or 'kill' the daemon. If it is to be started, then the script is called with 'start' as a parameter. If it is to be killed, then the script is called with 'stop' as a parameter. A valid list of parameters for each script will be printed if one calls the script by itself.

** If one wishes to temporarily disable a script at a particular runlevel, they can simply rename the link so that it does not start with an 'S' or a 'K'

When entering a runlevel, all the kill or stop scripts are run first before processing the start scripts. On bootup, things are a bit different. Because we have not come from another runlevel, all the kill or stop scripts are skipped. There is no need in stopping services that cannot be running in the first place. It should also be noted that a kill or stop script and a start script can exist for the same service in the same runlevel. This would cause the service to be restarted upon entering that runlevel.

Immediately following the `<action>` identifier is a two digit number that indicates the order it is to be executed. The scripts are actually run in alphabetical order, so the lower numbered scripts are run first.

The name that follows the two digit number isn't necessary, but helps greatly in identifying the links.

Finishing up

Once init finishes with the startup scripts, it runs a getty process on each terminal specified in inittab. Getty is what you see when you are logging in. Once you enter your username, the getty process runs login, which in turn asks for your password.

Getty doesn't have to be run here though. It could instead be replaced with something else. If one wanted to build an Internet kiosk, they could replace it with a script that started XFree86 and then a web browser like Mozilla.

There are many ways in which one might want to change the boot behaviour and there can sometimes be many chances for speeding up this process. Having a good understanding of the Linux boot process will greatly aid in ones ability to effectively maintain it.

If anyone wishes for a truly in-depth look at the boot process from the viewpoint of the programmer, they should check out "Linux Kernel 2.4 Internals" which is in the Guides section at the Linux Documentation Project [http://www.tldp.org/].

Notes

References used for this document include:

"From Power Up To Bash Prompt" -
[http://axiom.anu.edu.au/~okeefe/p2b/]

"Understanding the Linux Kernel" -
[http://www.oreilly.com/catalog/linuxkernel/]

"Linux Kernel 2.4 Internals"
[http://www.tldp.org/LDP/lki/index.html]

"README" file provided with LILO (extremely in-depth)
[http://lilo.go.dyndns.org/]

Linux Standard Base
[http://www.linuxbase.org/]

"x86 modes" -
[http://www.deinmeister.de/x86modes.htm]