

# An Introduction to GNU/Linux

## SESSION 2

Amr Ramadan

[amr.ramadan@gmail.com](mailto:amr.ramadan@gmail.com)

### The First Footstep: Booting

---

Once the BIOS finishes the preliminary operations required for initializing the computer hardware, such as initializing the registers, power management and performing the Power-On Self-Test, the BIOS then selects the device to boot from, whether that device is one of the hard drives in the computer, a USB drive, a floppy drive, or whatever.

After selecting the device, and let us assume it was some attached hard drive, the BIOS transfers control to the code residing in the Master Boot Record (MBR) of that device. The MBR is the first sector on a hard drive, in cylinder 0, head 0, sector 1, and is 512 bytes in size. In a properly configured system, the Master Boot Record should contain a Bootstrap program<sup>1</sup>, a program that begins the actual initialization of the operating systems.

---

<sup>1</sup> Although the Master Boot Record is 512 bytes in IBM PCs, not all of it contains executable code. The partition table for the four Primary partitions is also stored on the MBR and occupies 64 bytes. There are other reserved bytes for flags and signatures.

## **Bootstrapping from the MBR**

A bootstrap program is a program that has been compiled into machine language, and then installed in the boot sector. To run it, the code must be first loaded into memory, and then the first instruction in the code is executed. After the BIOS selects the device to boot from, it loads its first sector into memory, specifically at address 0000:7C00, and then transfers execution to the first instruction at this address.

To Demonstrate an example, consider the following assembly program:

```
begin:
    mov    AH, 0x0E        ; Function to print a character to the screen
    mov    AL, 'H'         ; character to print: E
    int    0x10            ; print the character
    mov    AL, 'i'         ; character to print: i
    int    0x10            ; print the character

loop_:
    jmp    loop_           ; loop forever.
```

The code simply prints to the screen the word Hi and loops on forever. If we were to assemble this code, and write it in the MBR, once the BIOS boots up it will load this program to memory and execute it.

A boot loader does not have the full functionality of an operating system. It is there because it is not possible to load the operating system in one shot. The boot loader, stored in the MBR, is responsible for selecting the appropriate partition to boot from. It does that by selecting the Primary partition marked as Active, and transfers control to the code in its Boot Sector. The partition's Boot Sector is similar to the MBR except that it is specific to a partition, not the whole disk.

Common bootstrap programs, also known as boot loaders, include GRUB (Grand Unified Boot Loader), LILO (Linux Loader) and NTLDR (NT Loader).

However, a little problem arises: 512 bytes may not even be enough to store the boot loader. Remember that the computer now has next-to-nothing functionality. For example, it does not understand a file system like FAT32. But then, if our boot loader supported FAT32, then what about an operating system stored on a partition formatted with another file system, say ext2fs<sup>2</sup>?

From here, it becomes apparent that the more universal the boot loader has to be, the bigger its code is. Yet, we are still limited to a 512-byte MBR, and there is no way to instruct the BIOS to load any more bytes.

The solution to this problem is to load the boot loader in multiple stages. The first stage would be stored in the Master Boot Record, which would then be only responsible for loading the next stage, and so on, till the operating system is loaded.

To take an example of multiple-stage boot loaders, consider GRUB, currently the most popular boot loader for GNU/Linux. GRUB is a multi-stage boot loader, and operates through either two or three stages. Stage 1 of GRUB is small enough to be installed in either the MBR or the partition's boot sector, and is responsible for loading either Stage 1.5 or Stage 2 (both are stored as files on the hard drive). Stage 1 stores the block address of Stage 1.5 or Stage 2. In other words, Stage 1 is still unaware of file systems;

---

<sup>2</sup> ext2fs is the Second Extended File System, one of the most popular file systems in use with the Linux kernel, and has only been replaced by ext3.

there is no way it can possibly interpret the file system structure and thus the block address of the next Stage has to be hard coded.

## **GRUB Stages**

---

Before GRUB is installed on a hard drive, it consists of a set of files, each file is known as an image. An image is a file containing a perfect copy of data that is to be later loaded as is. A Stage 1 image contains the compiled machine language code, and when we say an image is loaded into memory, this means this machine code is copied as is to the memory, so that it can be executed directly.

GRUB, before being installed on the hard drive, consists of a set of images. An image for Stage 1, an image for Stage 2 and one or more images for Stage 1.5.

These images are usually stored in `/boot/grub`. For example, this is the contents of the `/boot/grub` directory in a typical Redhat 9 installation:

```
7840 bytes  e2fs_stage1_5
7536 bytes  fat_stage1_5
6880 bytes  ffs_stage1_5
8448 bytes  jfs_stage1_5
7040 bytes  minix_stage1_5
9408 bytes  reiserfs_stage1_5
 512 bytes  stage1
106364 bytes stage2
 6528 bytes vstafs_stage1_5
 9320 bytes xfs_stage1_5
```

Now the question is, if Stage 1 is too big to fit in the MBR, and thus we stored in it the address of the next Stage, why do we need Stage 1.5? Why doesn't Stage 1 immediately load Stage 2? The answer is: it is possible. Stage 1 can directly load Stage 2. The difference between Stage 1 and Stage 1.5 is that the former does not understand any file system while the latter understands one filesystem (e.g. `e2fs_stage1_5` understands the Extended 2 filesystem). So you can move the Stage 2 image to another location safely, even after GRUB has been installed.

Now that GRUB is fully loaded into memory, we can use it to boot an operating system. Before we delve into what GRUB does exactly, we need to explain the concept of a kernel.

## **The Linux Kernel**

---

The Linux kernel is the part of the operating system that abstracts the underlying computer hardware from running programs. The Linux kernel consists of two main parts: the part of the kernel that does basic system services like multitasking, virtual memory, memory management, interrupt handling and file system support; and the device drivers which provide support for hardware devices.

But which device drivers are included in the kernel? There are certainly countless hardware devices, and it is difficult to imagine that drivers for all these devices would be included in the kernel. If they were, that would be quite a large kernel.

An option is to require all users to compile their own kernels. Grab the kernel source, select which device drives you want (for example, the device driver for an nVidia video card rather than an ATI video card), and compile the kernel. This would be the perfect solution, and it will certainly have the best performance.

However, compiling a kernel is not only time consuming, but far from being an easy method to ask novice users to do. For that, the Linux kernel supports what is termed loadable modules, a mean by which device drivers may be loaded at runtime without having to be included in the kernel.

That way, when you install some GNU/Linux distribution, let us says Fedora Core, the kernel image that is copied to your hard drive during installation is a pre-compiled image; this image contains the drivers that are most common - drivers like those that support Serial Ports or Keyboards. Other drivers, like a driver for a TV tuner card, could be loaded at runtime.

### **Loadable Modules**

The loadable modules are usually stored in `/lib/modules`, and can be loaded into memory by the programs `insmod` OR `modprobe`.

The kernel is always stored in the hard drive as an image file, usually in the `/boot` directory and is usually called `vmlinux` (uncompressed version), `vmlinuz` (compressed version), or a similar filename with a kernel version number at the end, for example `/boot/vmlinuz-2.6.13`. It is the boot loader's responsibility to load this file and transfer execution to it.

Where is the kernel loaded in memory? The memory is divided into two areas: the kernel memory (also known as the kernel space), and the user memory (also known as the user space). The kernel memory contains the kernel binary, and the working tables; a set of tables used to keep track of the status of the system and buffers. The user memory stores the users' programs.

### **Versioning Linux**

The Linux kernel's version consists of four numbers, in the form A.B.C[.D]. Bugfixes and security patches lead to a change in the D number, for example, a security vulnerability in kernel 2.6.13.3 could be fixed in 2.6.13.4. The C number changes when new features are introduced, like supporting a new file system, while the B number indicates a major revision of the kernel. The A number has changed only twice (from 0.99.11 to 1.0, and from 1.2.0 to 2.0.0), and is meant to be changed only when a major change in the code or concept occurs.

## **Back to GRUB**

---

What we have done so far is that we loaded GRUB into memory, nothing else! We have not loaded anything but GRUB and we have not yet prepared any partitions to read from them. What we need to do now is to instruct GRUB to load the kernel, and transfer execution to it.

As we described above, the kernel is stored in the hard drive as a binary image file, containing the compiled code of the kernel. As far as we understand so far, GRUB Stage 2 can understand file systems, so all we have to do is to tell GRUB to load a kernel image from a specified partition.

An example GRUB command to boot the system would be:

```
grub> root (hd0, 0)
grub> kernel /vmlinuz-i686-6.2.13.5 root=/dev/hda3
grub> boot
```

The first command tells GRUB that the partition containing the Kernel image is (hd0,0), which is the first partition in the Primary Master IDE hard disk drive. GRUB will try to mount the file system of this partition. If it supports the file system and succeeded in mounting, it goes on to the second command, which now that the partition was mounted and ready to be read from, instructs GRUB to load the kernel image named vmlinuz-i686-6.2.13.5, and passes to the kernel the argument root=/dev/hda3.

As you may have noticed, the first command describes a partition in a nomenclature different from that used in the second command. Since GRUB is designed to boot from a multitude of operating systems, it has to be neutral to the way a partition is described. As an example, the partition Slackware would name as /dev/hda2 would be referred to as /dev/ad1s1a if it was the FreeBSD operating system installed. To put in other words, each operating system describes a partition in a different way, and GRUB has to be independent from this. For the above example, GRUB is instructed to load from the kernel from the first hard disk (referred to as hd0), from the first partition in it (referred to as 0).

Also worth noticing is the distinction between the partition the kernel image is loaded from, and the root partition which is supposed to include the installed system. This distinction, as we shall see later with initrd images, allows for storing the system on a file system not understood by the boot loader.

After GRUB loads the kernel into the beginning of memory, it transfers execution to it. The kernel then decompresses itself (if the image was compressed), assigns the kernel memory and user memory, loads the drivers compiled inside of it and each driver initializes itself, mounts the root file system (passed as an argument. Above, it is /dev/hda3), and then executes /sbin/init.

### **Decompresses itself?**

We said that the Linux kernel may be stored on the hard drive as either a compressed or an uncompressed image, and that when it is loaded into memory by the boot loader, it decompresses itself. How does this work!?

The boot loader does not understand the contents of the kernel image. All it does is read it, load it in memory and transfer execution to the first instruction in it. If the kernel was compressed, then the image must contain in the beginning code to decompress the image. That is, the image consists of a compiled decompression routine followed by the actual kernel, compressed.

If the kernel was compressed by the gzip algorithm, then the image file will contain code to decompress the gzip file at its start. If it was compressed by the bzip2 algorithm, then the image file will contain code to decompress the bzip2 file at its start.

## Initial RAM Disk

---

In the above description of the boot process, we said that after the kernel was loaded, it executed the program `/sbin/init`. Here we made the assumption that the loaded kernel contains enough drivers that can read the disk drive and the file system containing the `init` program.

So, what if the `init` program was stored in a SCSI disk and the kernel does not have SCSI drivers? Or what if the `init` program was stored on a ReiserFS partition and the drivers for this file system were not loaded?

The solution to this problem is an `initrd`, the Initial RAM Disk. When using `initrd`, the boot loader will load an `initrd` image from the boot partition, and when the kernel starts executing, it will mount it in RAM as a RAM disk. By executing executables contained within the cached RAM disk, it loads up initial necessary drivers for a file system to operate. During this process, devices like USB devices, network cards, RAID arrays and others will get mounted so that the phase two process can boot up the operating system normally.

Note that it is possible to boot Linux without using an `initrd`; if nothing needs to be loaded before the file system is mounted (i.e. the device on which the root file system reside does not need external drivers to be loaded). This is usually the case for IDE based systems. In this case, it simply mounts the real file system and then configures the devices as normal. On SCSI systems, special drivers may be necessary, in which case the system will not boot without an `initrd` image to load them.

To specify a specific `initrd` image to GRUB, issue the following command

```
grub> initrd /initrd-i686.img
```

As of this moment we have started the boot loader, GRUB, which loaded into memory the kernel image, and transferred execution to its start. The code at the start of the kernel image decompresses itself, initializes the memory, loads and initializes the drivers compiled inside it, mounts the root file system, and then executes the `init` program.

## And Here Comes “init”

---

### Playing Around

---

After the kernel loads and does the required initialization, we said it executes the `init` program. To override this behavior, pass the parameter `init=/sbin/sh` to the kernel command in GRUB.

```
grub> kernel /vmlinuz-i686-6.2.13.5 root=/dev/hda3 init=/sbin/sh
```

This way, the kernel loads the program `/bin/sh` rather than the default program `/sbin/init`, so you go to the shell immediately.

`init` is the only program the kernel starts. All the other programs are started by `init`, and the sequence in which `init` starts programs varies from one distribution to the other. Before describing how `init` does what it needs to do, we need to describe Runlevels first.

A Runlevel is a mode of operation. It is the state of the system. A system that is running in Runlevel 3 is said to have a different state than the same system running in Runlevel 4. As an example, in most GNU/Linux distributions, Runlevel 1 is Single-User mode; Run Level 2 is Multi-User mode; Runlevel 3 is Multi-User Mode with the Network interfaces configured; and Runlevel 5 is similar to Runlevel 3 with the added support for the X Window System; Runlevel 6 reboots the system.

The role of `init` is to start the system in some runlevel, and this begins by `init` executing the activities defined in the file `/etc/inittab`. The consequence of executing this file is a set of commands being run. The specific commands, and the sequence in which they are run, depends on the selected run level. This process leads to initializing much of the operating system like the remaining partitions, loadable modules, set the system clock, set the hostname, and various other initialization tasks.

The last command executed through `/etc/inittab` is to start up the X-Window environment, in case of run level 5.

## References / Additional Readings

---

- NASM, the Netwide Assembler, if you want to assemble some code  
<http://sourceforge.net/projects/nasm>
- Dual Booting Tutorial, install Windows and then GNU/Linux  
[http://www.enterisedt.com/publications/dual\\_boot.html/](http://www.enterisedt.com/publications/dual_boot.html/)
- Filesystem Hierarchy Standard  
[http://en.wikipedia.org/wiki/Filesystem\\_Hierarchy\\_Standard](http://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard)
- GRUB Manual  
<http://www.gnu.org/software/grub/manual/grub.html>
- More details on `/etc/inittab`  
<http://www.pycs.net/lateral/stories/23.html#id7>
- Details of many versions of MBR and OS boot records  
[http://mirror.href.com/thestarman/asm/mbr/MBR\\_in\\_detail.htm](http://mirror.href.com/thestarman/asm/mbr/MBR_in_detail.htm)