Wake Up Computer!

or

The Boot Process

Orr Dunkelman

orrd@vipe.technion.ac.il

A Bit on the Boot Up Sequence

When you power on the computer it starts a sequence of operations which ends in a login window. Actually the login window is a process that the operating system has spawned.

The process of loading the operating system into the memory is very complicated and tricky. We start with some hardware boot handler, which passes the control to a software handler and end with a fully functional operating system.

Why Do We Need a Software Boot Handler?

If the hardware boot handler is good enough to pass control to the software handler, why it is not used to pass control directly to the operating system?

Boot Process – Theory Before Practice

Let us assume that we have a copy of the operating system ready to be loaded into the main memory. In Linux we call this file *kernel image*, in DOS it is the *MSDOS.SYS* and *IO.SYS*, and each and every operating system has this kind of file(s).

When the computer is off, this file is stored on the hard-drive. The hard-drive stores the files according to some file system. To read this file system we need the operating system's "read from hard-disk" functions.

These functions are available in the kernel image file, which can be accessed as soon as the image is loaded into memory...

To solve this endless loop, we use the concept of *bootstrapping*.

Bootstrapping

Bootstrapping is the art of using oneself to pull oneself out of the mud.

Let us assume that we start with a small place where all data is stored in pre-determined universally known format. We put in this small place a small program which knows how to access the hard-disk itself and load some additional program. This additional program can read the file-system and load files into the memory. And so it does: it reads the kernel image and writes it into the memory. Then this program hands over the computer to the kernel image which is a functional operating system.

Boot Process for an Intel Processor

For x86 Intel processors, powering them on (or pressing RESET) will cause the BIOS (Basic Input/Output System) to perform the POST (see later). After the POST is finished the operation *jmp ADDR* is loaded into the CPU, where *ADDR* is a constant address, which is '0x07C000'.

The BIOS can control keyboards, screens, disk drivers and several hardware interfaces (parallel and serial ports or nowadays also USB).

The BIOS searches in all the partitions it can access (according to a pre-determined order) for a boot sector (boot sector has a special value in a specific address).

The Role of the BIOS

The BIOS has 3 (very) important and major purposes:

- POST - Power On Self Test. This step is just to make sure that everything is fine. Some of the errors can cause the system to halt, and some of them are only reported.
- OS Loader - The BIOS is programmed to scan all possible boot devices (floppy drives, hard-disks, CD-Rom's and even network cards) to look for a boot program.
- Basic I/O Services - the BIOS can be used to access the system's basic hardware. Linux usually uses the BIOS during the initialization, but afterwards prefers to use its own handlers.

The OS Loading – First Phase

After the BIOS finds an active boot record (the place where the boot sector contains a OS-loader) it reads the boot sector and places it at 0x07C00. Boot records have 512 bytes. 446 of these are the loader itself, 64 bytes are the partition table, and the last 2 bytes are the signature of a boot record (0x55AA).

There are various boot loaders. We will discuss 3 boot loaders: the NT Loader, the LILO (linux loader) and GRUB. Each of these has his own unique features and concepts used.

The NT Loader

The NT Loader, as its name suggests is responsible for loading Microsoft Windows OSes into memory.

The NT Loader is installed when you install Microsoft Windows. It resides in the MBR (master boot record). Unlike Linux installations, installing Windows automatically installs the NT loader. However, unlike the Microsoft way, the previous contents of the MBR are kept in the file *BOOTSECT.DOS*.

The small MBR utility calls NTLDR, a small program which reside on the active partition's root directory (C: in most of the cases). The NTLDR program outputs the message "OS Loader V4.xx", announcing its existence to the user. Then it access a configuration file named BOOT.INI.

The NT Loader (cont.)

BOOT.INI example:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT

[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="NT V4 is here"
multi(0)disk(0)rdisk(0)partition(2)\WINNT="NT V4 VGAMODE" /basevideo /sos
C:\="DOS is here"
```

Once a Windows OS is asked for, the NT loader loads a program called NTDETECT.COM. This program checks that the existent hardware is OK, and if everything is fine – loads the NT kernel.

The question is what happens once the system is not Windows.

The NT Loader (cont.)

In that case the NTLDR program needs to know who is the true loader of the OS and to pass control to it. The mechanism is based on keeping an alternative MBR on the hard-disk in a file. For example, BOOTSECT.DOS which is automatically generated during installation.

So to have also Linux in the list of options, we just need to change the file BOOT.INI a little bit and add the line

```
C:\BOOTSECT.LIN="Now Linux is here"
```

at the end of the file (actually in the [operating systems] part).

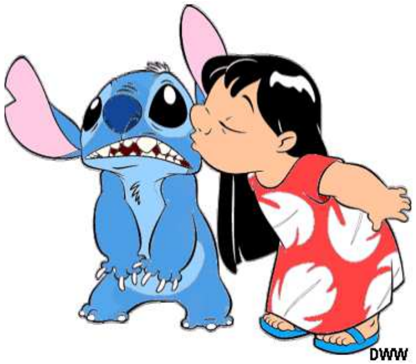
So we just need to create this file... and Voilá –
Linux from inside NT Loader. Later we will see how to create this file.

LILO

LILO

LILO is one of the first Linux loaders. Since its introduction in 1992 (by Werner Almesberger) LILO was used to control the Master Boot Record. LILO can be viewed as 3 utilities all called LILO:

1. LILO1 - The first boot loader.
2. LILO2 - The secondary boot loader.
3. LILO3 (Stitch) - The utility which updates LILO2's configuration.



LILO1, LILO2 and Stitch

The reason we need Stitch is due to the way LILO behaves. LILO1 and LILO2 once are loaded, do not try to scan the system (not even the Linux system) for any configuration files or information. All the configuration LILO (1 or 2) need must be hard-coded into them.

Stitch is run by *root* before the next boot. The root runs a utility called *lilo* which reads its configuration file (default - */etc/lilo.conf*), parses the file, and updates the tables in the hard-coded LILO2.

LILO1, LILO2 and Stitch (cont.)

LILO2 can contain up to 16 different entries. Each entry contains an option for which kernel to load, with the related parameters. In the case of non-Linux operating systems, the entry stores information about where that OS loader resides.

Once LILO1 is loaded to address 0x07C00 by the BIOS, it copies itself to the address 0x96A00 and sets a stack (0x96A00 downto 0x96800). It also loads the secondary boot loader (LILO2) at address 0x96C00 and transfers the control to it.

The OS Loading – Second Phase

The secondary boot loader loads the descriptor table at 0x98800 (which kernels are available), and waits for the user's input. The user has to choose which of the possible images he wants to load.

Actually these 16 images might contain a direction to a secondary loader of a non-Linux system like Solaris, MacOS, FreeBSD, or even Microsoft operating systems (NT Loader entries as well!).

When choosing the image to be used, we can also choose to pass parameters to the kernel.

The OS Loading – Second Phase (part II)

After choosing which kernel (and operating system) to load, the LILO2 loads some setup information at 0x90000 and 0x90200. Then after everything is ready for the kernel itself, LILO2 uncompresses the kernel image into address 0x10000. This address might change to 0x100000 if the kernel is compiled as "high".

If the loaded image is a Linux kernel image then the control is transfered to the setup code (0x90200). For other OSes the boot sequence is a bit more complex, but is essentially the same.

OS Loading – Second Phase – Reporting Errors

Murphy's Law: Anything that can go wrong will go wrong.

Like in any good program, errors will occur. The main problem which LILO faces is how to report the errors. Long error messages are out of the question. Even outputting an error code that was not returned by the BIOS is very limited.

The solution: LILO only announces success. If some stage fails, you don't get the success message.

If the first loader is not loaded, then the system halts, and no output is shown. After the successful loading, LILO1 outputs L.

OS Loading – Second Phase – Reporting Errors (2)

After LILO1 is loaded, it tries to load LILO2. If this step fails, it is usually problem with reading the media (either a media failure or bad geometry), and the error code returned by the BIOS is shown.

When the second loader is loaded, LILO1 which is still in charge outputs I. If the system is stuck at this point, it means that the first loader did not succeed to pass the control to the second loader.

This usually due to a mistake which caused LILO1 to load the wrong value. Mostly, these problems arises when LILO1 is not able to access LILO2 on the hard-disk (as in the past LILO1 could not access sectors of hard-disk whose number is greater than 1024).

OS Loading – Second Phase – Reporting Errors (3)

The ”LI” message is amongst the most hated ones in the Linux world (if not the most hated). In many cases it is equivalent to Blue Screen of Death (BSoD): You know you are in trouble, you have no idea why, and you cannot do nothing but restart your computer.

When LILO2 starts to run it outputs L. If the system halts at this point then the descriptor table is unreachable. If the second loader outputs L? it means that the loader was not loaded into the right place. L- means a corrupted descriptor table.

OS Loading – Second Phase – SUCCESS!

If everything went just fine then the second loader outputs the final O and waits to the user's selection of the kernel.

As mentioned before, once the kernel is chosen, the system loads it to the memory while uncompressing it. Then the control is given to the uncompressed kernel.

Question: Why does Linux compress its kernel?
The boot sequence is very complicated as is. Adding complexities by using compression seems very weird.

Compressing Kernel – A Big (small) Reason

In the system this lecture was written (Linux 2.4.18, 256 MB memory, dual Pentium 300 Mhz) the size of the compressed image file is 850 KB. The fully functional system (which supports SCSI RAID hard-drives, fully functional TCP connections and more) can be loaded onto a diskette!

This means you can get a fully equipped operating system that can be carried around on a diskette.

Another important reason is the x86 family design. When the processor starts to run it is in *real mode*. Thus, it can access only 1 MB of the memory (bits 20–31 of the addresses are all set to 0). If the kernel image is bigger than 1 MB, it requires special dealing.

Linux uses additional tricks to load itself (like using RAM disks), some of them are very useful for bypassing these restrictions.

OS Loading – First Phase Revisited

We can use other OS loaders as well. A very common loader nowadays is the GNU GRUB (hereandafter GRUB).

GRUB – Grab it While You Can

GRUB is based on the *Multiboot* methodology. The basic idea is to create a standard interface, which will allow any system to be loaded in any time.

GRUB is a very strong boot loader. It allows the user greater flexibility and much more options over both previously mentioned boot loaders.

The most important feature of GRUB is the fact that an error does not (necessarily) lock your system!

Amongst many other nice features, you can during the loading of the system replace the hard-disks map. This is equivalent to announcing that C: is D: and vice versa, whenever your boot requires that.

On the other hand, the OS must be able to load itself high (as Linux, and almost any modern OS can do).

GRUB – Grab it While You Can (Cont.)

Similarly to LILO and NT, a small part of GRUB (named *Stage 1*) resides in the MBR.

This part can either call *Stage 1.5* or *Stage 2*. The main difference between GRUB and LILO is mostly in *Stage 1.5*. *Stage 1.5* is a set of various GRUB utilities that can access file systems.

Those can be viewed as modules that are loaded by *Stage 1* when it needs to read the configuration file, look for the various kernel files, handle network boot, etc.

Stage 2 is quite similar to LILO2 after the kernel was selected. It is used mainly to load the chosen kernel, and pass control to it.

GRUB – How does it Work?

- GRUB *Stage 1* initialize the system (briefly).
- *Stage 1* detects the geometry and how to access the active drive.
- *Stage 1* loads the first sector of *Stage 1.5* or *Stage 2*.
- *Stage 1* passes control to the next stage.
- The next stage loads the rest of it, and starts executing it.

GRUB – Config Files

```
timeout 10
color black/cyan yellow/cyan
i18n (hd0,5)/boot/grub/messages
keytable (hd0,5)/boot/us-latin1.klt
altconfigfile (hd0,5)/boot/grub/menu.once
default 0

title linux
kernel (hd0,5)/boot/vmlinuz root=/dev/hda6 devfs=mount quiet vga=788
initrd (hd0,5)/boot/initrd.img

title linux-nonfb
kernel (hd0,5)/boot/vmlinuz root=/dev/hda6 devfs=mount
initrd (hd0,5)/boot/initrd.img

title failsafe
kernel (hd0,5)/boot/vmlinuz root=/dev/hda6 devfs=nomount failsafe
initrd (hd0,5)/boot/initrd.img

title NT
root (hd0,0)
makeactive
chainloader +1

title floppy
root (fd0)
chainloader +1
```

OS Loading – Third Phase

After the kernel gets the control it starts its initialization.

Basically this phase is quite similar for all OSes. The kernel sets its real-memory handling functions (which differ from the virtual memory handlers), the interrupt handlers (starting with the basic ones).

Then the kernel starts to load drivers. Some of them are part of the kernel and some can be used as modules (and thus can be replaced without compiling the entire kernel again).

The INIT Process – Creator of all Processes

In the beginning there was INIT, and the process table was empty, but the INIT. And the INIT said, Let there be daemons: and the daemons were loaded. And INIT saw the daemons, that they were good. And the INIT said, Let there be user front-ends. And INIT called the front-ends tty and pts.

INIT is the first process in all UNIX systems. As UNIX system requires that each process would have a father, INIT is the first in this tree of processes.

INIT loads all the services that the OS is supplying to the user (through several interfaces), and then loads the login screens[a].

[a] This can be avoided, if you don't really like it

References

[1] Linux Startup Process, Guy Keren, Haifux lecture 12, available on-line at http://www.haifux.org/lectures/12.

[2] Linux source code: init/main.c.

[3] LILO Technical Overview, version 21.

[4] Linux Documentation Project (http://www.ldp.org): From-PowerUP-To-Bash-Prompt HOWTO, LILO mini-HOWTO, Linux-Init HOWTO.

TODO: Finalize GRUB slides, Deal with hybernate, Update bibliography.