How your computer boots?

This article from Understanding the Linux Kernel by Daniel P. Bovet and Marco Cesati explains what happens right after users have switched on their computers, that is, how a Linux kernel image is copied into memory and executed. In short, we discuss how the kernel, and thus the whole system, is "bootstrapped."

Traditionally, the term *bootstrap* refers to a person who tries to stand up by pulling her own boots. In operating systems, the term denotes bringing at least a portion of the operating system into main memory and having the processor execute it. It also denotes the initialization of kernel data structures, the creation of some user processes, and the transfer of control to one of them.
Computer bootstrapping is a tedious, long task, since initially nearly every hardware device including the RAM is in a random, unpredictable state. Moreover, the bootstrap process is highly dependent on the computer architecture; as usual, we refer to IBM's PC architecture in this appendix.

Prehistoric Age: the BIOS

The moment after a computer is powered on, it is practically useless because the RAM chips contain random data and no operating system is running. To begin the boot, a special hardware circuit raises the logical value of the RESET pin of the CPU. After RESET is thus asserted, some registers of the processor (including cs and eip) are set to fixed values, and the code found at physical address 0xfffffff0 is executed. This address is mapped by the hardware to some read-only, persistent memory chip, a kind of memory often called ROM (read-only memory). The set of programs stored in ROM is traditionally called BIOS (Basic Input/Output System), since it includes several interrupt-driven low-level procedures used by some operating systems, including Microsoft's MS-DOS, to handle the hardware devices that make up the computer.
Once initialized, Linux does not make any use of BIOS but provides its own device driver for every hardware device on the computer. In fact, the BIOS procedures must be executed in real mode, while the kernel executes in protected mode, so they cannot share functions even if that would be beneficial.
BIOS uses real mode addresses because they are the only ones available when the computer is turned on. A real mode address is composed of a seg segment and an off offset; the corresponding physical address is given by seg×16+off. As a result, no global descriptor table (GDT), local descriptor table (LDT), or paging table is needed by the CPU addressing circuit to translate a logical address into a physical one. Clearly, the code that initializes the GDT, LDT, and paging tables must run in real mode.

Linux is forced to use BIOS in the bootstrapping phase when it must retrieve the kernel image from disk or from some other external device. The BIOS bootstrap procedure essentially performs the following four operations:

1. Executes a series of tests on the computer hardware, to establish which devices are present and whether they are working properly. This phase is often called POST (power-on self-test). During this phase, several messages, such as the BIOS version banner, are displayed.
2. Initializes the hardware devices. This phase is crucial in modern PCI-based architectures, since it guarantees that all hardware devices operate without conflicts on the IRQ lines and I/O ports. At the end of this phase, a table of installed PCI devices is displayed.
3. Searches for an operating system to boot. Depending on the BIOS setting, the procedure may try to access (in a predefined, customizable order) the first sector (*boot sector*) of any floppy disk, any hard disk, and any CD-ROM in the system.
4. As soon as a valid device is found, copies the contents of its first sector into RAM, starting from physical address 0x00007c00, then jumps into that address and executes the code just loaded.

The rest of this article takes you from the most primitive starting state to the full glory of a running Linux system.

Ancient Age: the boot loader

The *boot loader* is the program invoked by the BIOS to load the image of an operating system kernel into RAM. Let us briefly sketch how boot loaders work in IBM's PC architecture.

To boot from a floppy disk, the instructions stored in its first sector are loaded in RAM and executed; these instructions copy all the remaining sectors containing the kernel image into RAM.

Booting from a hard disk is done differently. The first sector of the hard disk, named the master boot record (MBR), includes the partition table and a small program, which loads the first sector of the partition containing the operating system to be started. Some operating systems, such as Microsoft Windows 98, identify this partition by means of an *active* flag included in the partition table; following this approach, only the operating system whose kernel image is stored in the active partition can be booted. As we shall see later, Linux is more flexible since it replaces the rudimentary program included in the MBR with a sophisticated program called LILO that allows users to select the operating system to be booted.

Booting Linux from floppy disk

The only way to store a Linux kernel on a single floppy disk is to compress the kernel image. As we shall see, compression is done at compile time and decompression by the loader.

If the Linux kernel is loaded from a floppy disk, the boot loader is quite simple. It is coded in the arch/i386/boot/bootsect.S assembly language file. When a new kernel image is produced by compiling the kernel source, the executable code yielded by this assembly language file is placed at the beginning of the kernel image file. Thus, it is very easy to produce a bootable floppy containing the Linux kernel. The floppy can be created by copying the kernel image starting from the first sector of the disk. When the BIOS loads the first sector of the floppy disk, it copies the code of the boot loader.

The boot loader, which is invoked by the BIOS by jumping to physical address 0x00007c00, performs the following operations:

1. Moves itself from address 0x00007c00 to address 0x00090000.
2. Sets up the real mode stack, from address 0x00003ff4. As usual, the stack will grow toward lower addresses.
3. Sets up the disk parameter table, used by the BIOS to handle the floppy device driver.
4. Invokes a BIOS procedure to display a "Loading" message.
5. Invokes a BIOS procedure to load the setup() code of the kernel image from the floppy disk and puts it in RAM starting from address 0x00090200.
6. Invokes a BIOS procedure to load the rest of the kernel image from the floppy disk and puts the image in RAM starting from either low address 0x00010000 (for small kernel images compiled with make zImage) or high address 0x00100000 (for big kernel images compiled with make bzImage). In the following discussion, we will say that the kernel image is "loaded low" or "loaded high" in RAM, respectively. Support for big kernel images was introduced quite recently: While it uses essentially the same booting scheme as the older one, it places data in different physical memory addresses to avoid problems with the ISA hole mentioned in the section "Reserved Page Frames" in Chapter 2.
7. Jumps to the setup() code.

Booting Linux from hard disk

In most cases, the Linux kernel is loaded from a hard disk, and a two-stage boot loader is required. The most used Linux boot loader on Intel systems is named LILO (LInux LOader); corresponding programs exist for other architectures. LILO may be installed either on the MBR, replacing the small program that loads the boot sector of the active partition, or in the boot sector of a (usually active) disk partition. In both cases, the result is the same: When the loader is executed at boot time, the user may choose which operating system to load.

The LILO boot loader is broken into two parts, since otherwise it would be too large to fit into the MBR. The MBR or the partition boot sector includes a small boot loader, which is loaded into RAM starting from address 0x00007c00 by the BIOS. This small program moves itself to the address 0x0009a000, sets up the real mode stack (ranging from 0x0009b000 to 0x0009a200), and loads the second part of the LILO boot loader into RAM starting from address 0x0009b000. In turn, this latter program reads a map of available operating systems from disk and offers the user a prompt, so she can choose one of them. Finally, after the user has chosen the kernel to be loaded (or let a time-out elapse so that LILO chooses a default), the boot loader may either copy the boot sector of the corresponding partition into RAM and execute it or directly copy the kernel image into RAM.

If a Linux kernel image must be booted, the LILO boot loader, which relies on BIOS routines, performs essentially the same operations as the boot loader integrated into the kernel image described in the previous section about floppy disks. The loader displays the "Loading Linux" message; then it copies the integrated boot loader of the kernel image to address 0x00090000, the setup() code to address 0x00090200, and the rest of the kernel image to address 0x00010000 or 0x00100000. Then it jumps to the setup() code.

Middle Ages: The setup() function

The code of the setup() assembly language function is placed by the linker immediately after the integrated boot loader of the kernel, that is, at offset 0x200 of the kernel image file. The boot loader can thus easily locate the code and copy it into RAM starting from physical address 0x00090200.

The setup() function must initialize the hardware devices in the computer and set up the environment for the execution of the kernel program. Although the BIOS already initialized most hardware devices, Linux does not rely on it but reinitializes the devices in its own manner to enhance portability and robustness. Essentially, setup() performs the following operations:

1. Invokes a BIOS procedure to find out the amount of RAM available in the system.
2. Sets the keyboard repeat delay and rate. (When the user keeps a key pressed past a certain amount of time, the keyboard device sends the corresponding keycode over and over to the CPU.)
3. Initializes the video adapter card.
4. Reinitializes the disk controller and determines the hard disk parameters.
5. Checks for an IBM Micro Channel bus (MCA).
6. Checks for a PS/2 pointing device (bus mouse).
7. Checks for Advanced Power Management (APM) BIOS support.
8. If the kernel image was loaded low in RAM (at physical address 0x00010000), moves it to physical address 0x00001000. Conversely, if the kernel image was loaded high in RAM, setup does not move it. This step is necessary because, to be able to store the kernel image on a floppy disk and to save time while booting, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM.
9. Sets up a provisional interrupt descriptor table (IDT) and a provisional global descriptor table (GDT).
10. Resets the floating-point unit (FPU), if any.
11. Reprograms the Programmable Interrupt Controller (PIC) and maps the 16 hardware interrupts (IRQ lines) to the range of vectors from 32 to 47. The kernel must perform this step because the BIOS erroneously maps the hardware interrupts in the range from 0 to 15, which is already used for CPU exceptions.
12. Switches the CPU from real mode to protected mode by setting the PE bit in the cr0 status register. As explained in the section "Kernel Page Tables" in Chapter 2, the provisional kernel page tables contained in swapper_pg_dir and pg0 identically map the linear addresses to the same physical addresses. Therefore, the transition from real mode to protected mode goes smoothly.
13. Jumps to the startup_32() assembly language function.

Renaissance: The startup_32() functions

There are two different startup_32() functions; the one we refer to here is coded in the arch/i386/boot/compressed/head.S file. After setup() terminates, the function has been moved either to physical address 0x00100000 or to physical address 0x00001000, depending on whether the kernel image was loaded high or low in RAM.

This function performs the following operations:

1. Initializes the segmentation registers and a provisional stack.
2. Fills the area of uninitialized data of the kernel identified by the _edata and _end symbols with zeros.
3. Invokes the decompress_kernel() function to decompress the kernel image. The "Uncompressing Linux . . . " message is displayed first. After the kernel image has been decompressed, the "O K, booting the kernel" message is shown. If the kernel image was loaded low, the decompressed kernel is placed at physical address 0x00100000. Otherwise, if the kernel image was loaded high, the decompressed kernel is placed in a temporary buffer located after the compressed image. The decompressed image is then moved into its final position, which starts at physical address 0x00100000.
4. Jumps to physical address 0x00100000.

The decompressed kernel image begins with another startup_32() function included in the arch/i386/kernel/head.S file. Using the same name for both the functions does not create any problems (besides confusing our readers), since both functions are executed by jumping to their initial physical addresses.

The second startup_32() function essentially sets up the execution environment for the first Linux process (process 0). The function performs the following operations:

1. Initializes the segmentation registers with their final values.
2. Sets up the kernel mode stack for process 0.
3. Invokes setup_idt() to fill the IDT with null interrupt handlers.
4. Puts the system parameters obtained from the BIOS and the parameters passed to the operating system into the first page frame.
5. Identifies the model of the processor.
6. Loads the gdtr and idtr registers with the addresses of the GDT and IDT tables.
7. Jumps to the start_kernel() function.

Modern Age: The start_kernel() function

The start_kernel() function completes the initialization of the Linux kernel. Nearly every kernel component is initialized by this function; we mention just a few of them:

- The page tables are initialized by invoking the paging_init() function.
- The page descriptors are initialized by the mem_init() function.
- The final initialization of the IDT is performed by invoking trap_init() and init_IRQ().
- The slab allocator is initialized by the mem_cache_init() and kmem_cache_sizes_init() functions.
- The system date and time are initialized by the time_init() function.
- The kernel thread for process 1 is created by invoking the kernel_thread() function. In turn, this kernel thread creates the other kernel threads and executes the /sbin/init program.

Besides the "Linux version 2.2.14…" message, which is displayed right after the beginning of start_kernel(), many other messages are displayed in this last phase, both by the init functions and by the kernel threads. At the end, the familiar login prompt appears on the console (or in the graphical screen if the X Window System is launched at startup), telling the user that the Linux kernel is up and running.