Booting the Kernel

Alessandro Rubini

rubini@linux.it

This article describes the steps that are performed to boot the Linux kernel. While this kind of information is not relevant to the system's functionality, it's interesting to see how the different architectures bring the system up.

1. Copyright and Licensing

2. Introduction

A computer system is a complex machinery, and the operating system is an elaborate tool that unrolls hardware complexities to end up showing a simple and standardized environment to the end user. When the power is turned on, however, the system software must work in a limited environment, and it must load the kernel using this scarse operating environment.

I'm going to describe here the booting process of three platforms: the old-fashioned PC and the more featured Alpha and Sparc platforms. The PC will take most of the space in this article because it is still more widespread than other platforms, and also because it's the most tricky platform to bring up. I am not going to show any code in this issue of the Kernel Korner because assembly is unintelligible to most readers, and each platform has its own assembly language.

3. The Computer at Power-On

In order to be able to do something with the computer when power is applied, things are arranged so that the processor begins execution from the system's firmware. The firmware is "unmovable software" found in ROM memory; some companies call it BIOS (Basic Input-Output System) to underline its software role, some call it PROM or "flash" to stress on its hardware implementation, while someone else calls it "console" to focus on user interaction.

The firmware usually checks that the hardware is correctly working, and retrieves part (or all) of the kernel from a storage medium and executes it. This first part of the kernel must load the rest of itself and initialize the whole system. I won't deal with firmware issues here, but only with kernel code, whose source is distributed along with Linux.

4. The PC

When the x86 processor is turned on in a personal computer, it is a 16-bit processor that only sees one meg of RAM. This environment is known as "real mode", and is dictated by compatibility with older processors of the same family. Everything that makes up a complete system must live within the available meg of address space: the firmware, video buffers, space for expansion boards and a little RAM (the infamous 640kB) must all be there.

To make things difficult, the PC firmware only loads half a kilobyte of code, and establishes its own memory layout before loading this first sector. Whichever the boot media, the first sector of the boot partition is loaded in memory to address 0x7c00, where execution begins. What happens at 0x7c00 depends on the boot-loader being used; I'm going to examine three situations here: no boot-loader, lilo, loadlin.

5. Booting zImage and bzImage

Even though it's pretty rare to boot the system without a boot loader, it is still possible to do so by copying the raw kernel to a floppy disk. A command like cat zImage > /dev/fd0[/] will work perfectly on Linux, although some other Unix systems can do the task reliably only by using the dd command. The raw floppy image thus created can then be configured by using the rdev program, but I won't discuss it here.

The file called zImage is the compressed kernel image that lives in arch/i386/boot after you issued make zImage or make boot – the latter invocation is the one I prefer, as it works unchanged on other platforms. If you built a big zImage, instead, the file is called bzImage, and lives in the same directory.

Booting an x86 kernel is a tricky task because of the limited amount of available memory. The Linux kernel tries to maximize usage of the low 640 kilobytes by moving itself around several times. But let's see in detail the steps performed by a zImage kernel; all the following pathnames are relative to arch/i386/boot.

The first sector (executing at 0x7c00) moves itself to 0x90000 and loads subsequent sectors after itself, getting them from the boot device using the firmware's funtions to access the disk. The rest of the kernel is then loaded to address 0x10000, allowing for a maximum size of half a meg of data – but this is the compressed image. The boot-sector code lives in bootsect.S, a real-mode assembly file.

Then, code at 0x90200 (defined in setup.S) takes care of some hardware initialization and allows to change the default text mode (video.S). Text mode selection has become a compile-time option from 2.1.9 onwards.

Later, all the kernel is moved from 0x10000 (64K) to 0x1000 (4K). This move overwrites BIOS data stored in RAM, and no BIOS call can be performed after then. The first physical page is not touched because it is the so-called "zero-page", used in handling virtual memory.

At this point setup.S enters protected mode and jumps to 0x1000, where the kernel lives. All the available memory can be accessed now, and the system can begin to run.

The steps just shown used to be the whole story of booting when the kernel was small enough to fit half a meg – the address range between 0x10000 and 0x90000. When the kernel was small it lived at 0x1000, but as features were added to the system it didn't fit half a meg any more: code at 0x1000 isn't the Linux kernel nowadays, but rather the "gunzip" part of the gzip program. The following additional steps are needed to uncompress the real kernel and execute it:

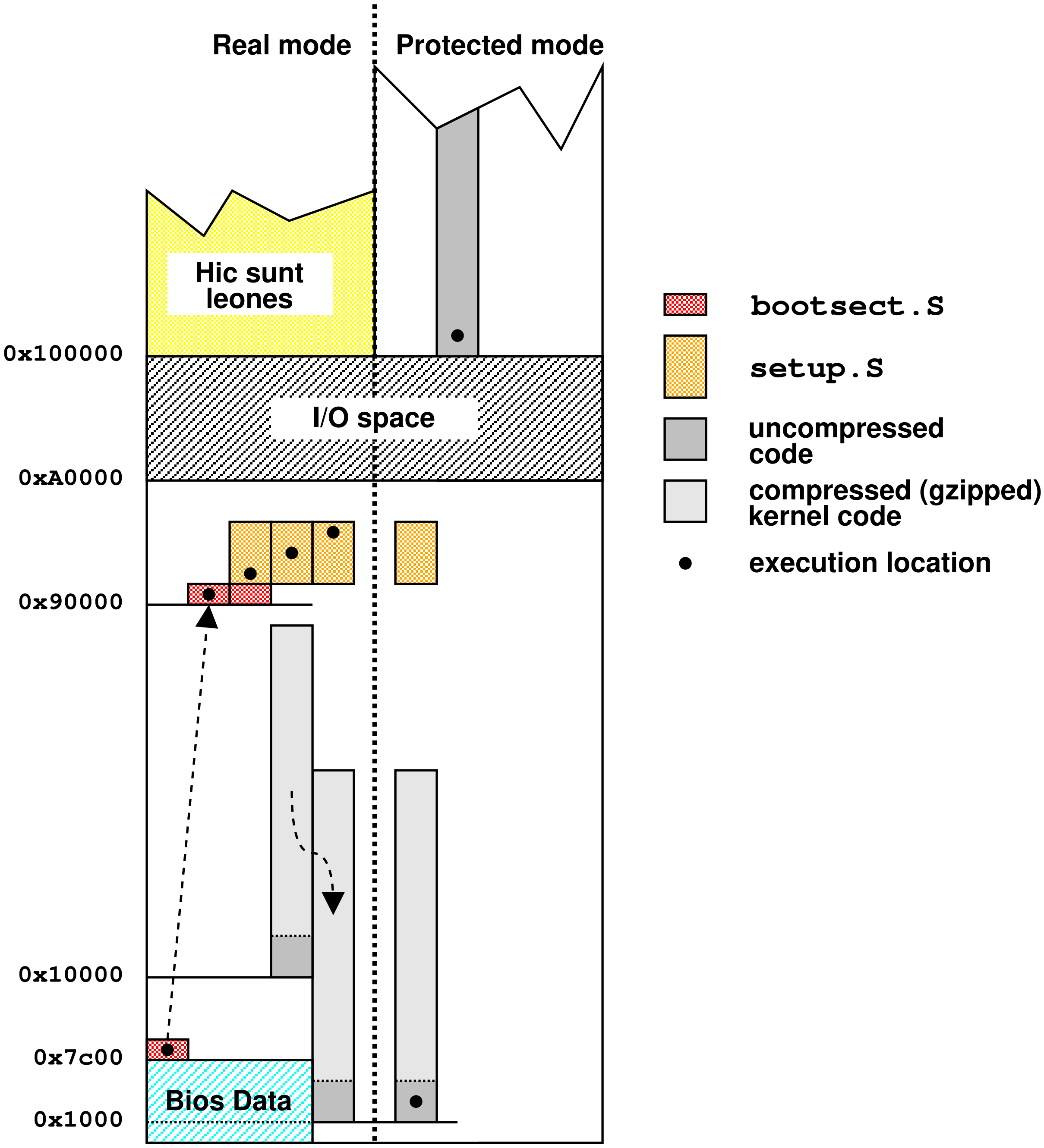
Code at 0x1000 is compressed/head.S, and is in charge of "gunzipping" the kernel: it calls the function decompress_kernel, defined in compressed/misc.c, which in turns calls inflate which writes its output starting at address 0x100000 (one meg). High memory can now be accessed, because the processor is definitely out of its limited boot environment – the "real" mode.

After decompression, head.S jumps to the real beginning of the kernel. The relevant code is in ../kernel/head.S, outside of the boot directory.

Boot is over now, and head.S (i.e., the code found at 0x100000 that used to be at 0x1000 before introducing compressed boots) can complete processor initialization and call start_kernel(). Everything is written in C from now on.

The various data movements that are performed at system boot are depicted in Figure 1.

Figure 1. Data Movements Performed at System Boot



The boot steps shown up to now rely on the assumption that the compressed kernel can fit in half a meg of space. While the assumption holds most of the times, a system stuffed of device drivers might not fit any more. This oversizing may happen for example to kernels used in installation disks: these kernels can easily get bigger than the available space, and some new machinery is needed to fix the problem. This something is called bzImage, and has been introduced in kernel version 1.3.73.

A bzImage is generated by issuing make bzImage from the toplevel Linux source directory. This kind of kernel image boots similarly to the zImage, with a few changes:

- When the system is loaded to address 0x10000, a little helper routine is called after loading each 64k data block. The helper routing moves the data block to high memory by using a special BIOS call. Only not-so-old BIOS'es implement the functionality, and that's why make boot still builds the conventional zImage as I write this article – but this might change in the near future.
- setup.S doesn't move the system back to 0x1000 (4k), but jumps instead directly to address 0x100000 (one meg) after entering protected mode. "One meg" is where data has been moved by the BIOS in the previous step.
- The decompressor found at one-meg writes the uncompressed kernel image in low memory until it gets exhausted, and then in high memory after the compressed image. The two pieces are then reassembled to address 0x100000 (one meg). Several memory moves are needed to perform the task correctly, but I won't detail the issue any deeper.

The rule for building the big compressed image can be read from Makefile: it affects several files in arch/i386/boot. One good point of bzImage is that when kernel/head.S gets called it won't notice the extra work, and everything will go on as usual.

6. Using Lilo

Most Linux-x86 users don't boot the raw kernel image from a floppy, but rather boot Lilo from the hard disk. Lilo replaces part of the process outlined above so that it can load a Linux kernel that is scattered throughout a disk. This allows the user to boot a kernel file off a filesystem partition, without using the floppy.

In practice, Lilo uses the BIOS services to load single sectors from the disk, and then jumps to setup.S. In other words, it arranges the memory layout like bootsect.S does, so the usual booting mechanism can complete painlessly. Lilo is also able to handle a kernel command line, and this is a good reason by itself to avoid booting the raw kernel image.

If you want to boot a bzImage with Lilo, you need version 18 or newer of the tool. Earlier versions of Lilo are not able to load segments to high memory, which is needed when loading big images in order for setup.S to find the expected memory layout.

The main disadvantage of Lilo is that is uses the BIOS to load the system. This forces to put the kernel and other relevant files in disks that can be accessed by the BIOS, and in the first 1024 cylinders of them. Actually, when you use the PC firmware you really discover how old-fashioned the architecture is.

Even if you don't run Lilo, you can enjoy the documentation files that are distributed with Lilo's source code. They document the boot process on the PC, and explain how to handle (almost) every conceivable situation.

7. Using Loadlin

If you want to boot your Operating System (uppercase) off another operating system (lowercase), Loadlin is the tool for you. The program is similar to Lilo because it loads the kernel from a disk partition and then jumps to setup.S. It is different from Lilo in that is must not only face the BIOS restrictions, but also get rid of an established memory layout without compromising the system' stability. On the other hand, it is not restricted to be half-a-kilobyte long, because it is not a boot sector but a complete program file.

> NOTE: Version 1.6 and newer of the program are able to load big images.

Loadlin is able to pass a command line to the kernel and is therefore as flexible as Lilo; most of the times you'll end up writing a linux.bat file to pass a full-featured command line to Loadlin when calling the linux command.

You can use Loadlin to turn any networked PC into a Linux box: you only need a kernel image equipped for mounting the root partition via NFS, Loadlin and a linux.bat with the correct IP numbers in. Sure you need a properly configured NFS server as well, but any Linux machine can do the job. For example, the following command line turns my gilfriend's PC (alfred.unipv.it) into a workstation:

Well, it must be properly formatted for your system....

@cx:loadlin c:\zimage rw nfsroot=/usr/root/alfred nfsaddrs=193.204.35.117:193.204.35.110:193.204.35.254:255.255.255.0:alfred.unipv.it

8. More of it

As you might imagine, the code is not as easy as I described it: it must deal with a lot of details, like bringing around the kernel's command line, keep an eye over the boot technique being used and so on. The curious reader can look in the source file to learn something more and to read the authors' comments that live herein. There's a lot of information in the comments, and they are often funny to read.

I personally don't feel you'll ever need to touch the boot code, because things get much more interesting when the system is up and running: you can exploit all the features of your processor and all the available RAM without getting mad with processor-level issues.

9. Booting an Alpha box

The Alpha platform is much more mature than the PC and its firmware reflects this maturity. My experience with Alpha is limited to the ARC firmware, which is anyway the most used.

After performing the usual detection of devices, the firmware displays a boot menu which lets you choose what file to boot. The firmware is able to read a disk partition (though only a FAT partition), so you actually boot a "file", without the need to hack boot sectors and build maps of disk blocks.

The file that gets booted will usually be linload.exe, which in turn loads Milo (the "Mini Loader", whose name is a pun about Milo's size). In order to boot Linux through the ARC firmware you need to have a small FAT partition on your hard drive to store linload.exe and milo. The Linux kernel doesn't need to access the partition unless you upgrade Milo, so FAT support can be left out of your Alpha kernel without incurring in side effects.

Actually, the user can exploit different options: the ARC boot menu can be configured to boot Linux by default, and Milo can even be burnt in flash memory in order to get rid of the FAT partition. But whatever you do, you end up with Milo running.

The Milo program is a stripped-down version of the Linux kernel: it has all the Linux device drivers and some filesystem decoder; unlike the kernel it doesn't have process control and includes Alpha initialization code. The tool is able to setup virtual memory and enable it, and can load a file from either an ext2 partition or an iso9660 device. The "file" in question is loaded to virtual address 0xfffffc0000300000 and then executed. The virtual address used is the one where the Linux kernel runs: it's unlikely you'll ever load anything but Linux, with the exception of the fmu (flash management utility) program used to burn Milo in flash ROM — fmu is compiled to execute from the same virtual address whence the kernel runs and is distributed with Milo.

It's interesting to note that Milo also includes a small 386 emulator and some of the PC BIOS functionality. This is needed in order to execute self-initialization code found on many ISA/PCI peripheral boards (PCI boards, though claiming to be processor-independent, use intel machine code in their ROM images).

But, if Milo does all of this, what is left to the Linux kernel?

A very little, actually. The first kernel code to execute in Linux-Alpha is arch/alpha/kernel/head.S, and it just needs to setup a few pointers and jump to start_kernel(). Actually, kernel/head.S for Alpha is much shorter than the equivalent x86 source file.

If you don't want to run Milo there is an alternative, though not a practical one. In arch/alpha/boot you'll find the sources of a "raw" loader which gets compiled by issuing make rawboot from the toplevel Linux source directory. The utility is able to load a file from a sequential region of a device (the floppy or the hard disk) using the firmware's callbacks.

In practice, the raw loader accomplishes a task similar to what bootsect.S does for the PC platform, and this forces to copy the kernel to either a raw floppy or a raw hard-disk partition. As you see, there's no real reason to try out this technique, which is quite hairy and lacks the flexibility Milo offers. I personally don't even know if it still works: the "PALcode" used by Linux is exported by Milo, and is different from the one exported by the ARC firmware. The PALcode is a library of low-level functions used by Alpha processors to implement low-level hardware management like paging; if the current PALcode implements different operations than the software expects, the system won't work.

10. Booting a Sparc station

Bringing up a Sparc computer is similar to booting the Alpha on the user side, and similar to booting the PC on the software side.

What the user sees it that the firmware loads a program and executes it, the program in turn is able to retrieve and uncompress a file found on a disk partition. The "program" in question is called Silo, and it can read files from either an ext2 partition or an ufs one. Unlikely Milo (likely Lilo), Silo is able to boot another operating system. There is no such need for the Alpha, because the firmware can boot multiple systems: once you run Milo, you have already made your choice – the Right Choice.

When a Sparc computer boots, the firmware loads a boot sector after performing all the hardware checks and device initialization. It's interesting to note that Sbus devices are platform independent, and their initialization code is portable Forth code rather than machine language bound to a particular processor.

The boot sector that gets loaded is what you find in /boot/first.b in your Linux-Sparc system, and is a bare 512 bytes. It is loaded to address 0x4000 and its role is retrieving from disk /boot/second.b and putting it to address 0x280000 (2.5 megs); the address has been chosen because the Sparc specifications state that at least three megabytes of RAM are mapped at boot time.

Everything else is then performed by the second-stage boot loader: it is linked with libext2.a to access system partitions, and can thus load a kernel image from your Linux filesystem. It can also uncompress the image because it includes inflate.c, from gzip.

second.b accesses a configuration file called /etc/Silo.conf, similar in shape to [cw]lilo.conf[/]. Since the file is read at boot time there's no need to re-install the kernel maps when a new kernel is added to the boot choices. When Silo shows its prompt you can choose from any kernel image (or other operating system) specified in Silo.conf, or you can specify a complete device/pathname pair to load a different kernel image without editing the configuration file.

Silo loads the disk file to address 0x4000. This means that the kernel must be shorter than 2.5 megs: if it is longer Silo will refure to overwrite its own image. No conceivable Linux-Sparc kernel is currently bigger than thant, unless you compiled it with -g to have debugging information available. In this case the kernel image must be stripped before being handled to Silo.

Finally, Silo performs kernel decompression and/or remapping to place the image at virtual address 0xf0004000. The code that takes over Silo is — as you may imagine — arch/sparc/kernel/head.S. The source includes all the trap tables for the processor and the actual code to set the machine up and call start_kernel(). The Sparc version of head.S is actually quite big.

11. start_kernel and on

After architecture-specific initialization is over, init/main.c takes control of the processor — whichever the processor is.

The start_kernel() function calls setup_arch() first, which is the last architecture-specific function. Unlike other code, however, setup_arch() can exploit all the processor's features, and is a much easier source file than the ones described earlier. The function is defined in kernel/setup.c under each architecture source tree.

The function then initializes all the kernel's subsystems — IPC, networking, buffer cache and so on. After all initialization is over, these two lines complete start_kernel():

@cx:
kernel_thread(init, NULL, 0);
cpu_idle(NULL);

The init thread is process number 1: it mounts the root partition, executes /linuxrc if CONFIG_INITRD has been selected at compile time, and then executes the init program. If init can't be found, /etc/rc is executed; using rc is discouraged nowadays, as init is much more flexible than a shell script in handling system configuration.

If neither init nor /etc/rc can be run, or if they exit, /bin/sh is executed repeatedly. This feature only exists as a safeguard in case the system administrator removes or corrupts init by mistake: if you remove a.out support from the kernel forgetting that your old init has not been recompiled, you'll enjoy having at least a shell running after reboot.

The kernel has nothing more to do after spawning process number 1, and everything else is handled in user space – by init, /etc/rc or /bin/sh.

And process 0? we've seen hoe the so called idle task executed cpu_idle(): this is a function that calls the idle() function in an endless loop. idle(), in turn is an architecture-dependent function, which is usually in charge of turning off the processor to save power and increase the processor's lifetime.