Bootloader

A Guide to getting familiar with The Bootloader

techw803@gmail.com, gauravraj.xyz, @thehackersbrain

2022-07-02

Contents

1 Bootloader **1**

1.1 Theory 1

1.1.1 What does a boot loader do 1

1.1.2 Loading your kernel 1

1.1.3 Giving the kernel its information 2

1.1.4 Establishing an environment 2

2 Bootloader design **3**

2.1 Single Stage Bootloader 3

2.2 Two-Stage Bootloader 3

2.3 Mixed Bootloader 3

3 Booting Multiple OSes **4**

1 Bootloader

> What a bootloader is?

1.1 Theory

A bootloader is a program written to load a more complex kernel. Implementation details are gathered in Rolling Your Own Bootloader

1.1.1 What does a boot loader do

The boot loader ultimately has to:

- Bring the kernel (and all the kernel needs to bootstrap) into memory
- Provide the kernel with the information it needs to work correctly
- Switch to an environment that the kernel will like
- Transfer control to the kernel.

On the x86, the boot loader runs in [Real Mode](https://wiki.osdev.org/Real_Mode. consequently it has easy access to BIOS resources and functions.Therefore it's a good place to perform memory map detection, detection of available video modes, loading of additional files, etc. The boot loader will collect this information and present it in a way the kernel will be able to understand.

1.1.2 Loading your kernel

The bits of your kernel are somewhere on some disk (presumably the booting disk, but this is not mandatory). Question is: where on the disk? It it a regular file on a FAT-formatted partition? It it a collection of consecutive sectors in the "reserved area" of the FAT file system (in which case you may need a dedicated tool to format the disk and install the kernel on it)? Or is the disk/partition simply left unformatted and the kernel pasted directly with a disk image tool?

All the above options are possible. Maybe the one I'd choose myself would be to reserve enough space on a FAT file system to store the list of sectors used by the kernel file. The field reserved sectors in the BPB is a perfect place for this. The "advantage" of being fully-FAT is that you don't need to re-write the bootsector every time you rewrite the kernel.

What needs to be loaded mainly depends on what's in your kernel. Linux, for instance, requires an additional 'initrd' file that will contain the 'initialization process' (as user level). If your kernel is modular and if Filesystems are understood by some modules, you need to load the modules along with the kernel. Same goes for 'microkernel services' like disk/files/memory services, etc.

1.1.3 Giving the kernel its information

Some kernels require some extra information to run. For example, you'll need to tell the Linux root partition to start from. Pretty useful information to have is a map of the address space - effectively a map where physical memory is and where it's not. Other popular queries regard video modes. In general, anything that involves a BIOS call is easier to do in Real Mode, so better do them while in real mode than trying to come back to real mode for a trip later.

1.1.4 Establishing an environment

Most kernels require protected mode. For these kernels you'll have to

- Enable A20
- Load a GDT
- Enter Protected mode

before giving control to the kernel.

It's common for the loader to keep interrupts disabled (the kernel will enable them later when an IDT is properly set up).

Note: *take time to thing about whether or not you'll enable paging here. Keep in mind that debugging paging initialization code without the help of exception handlers may quickly become a nightmare!*

2 Bootloader design

Virtually any bootloader follows a common design.

2.1 Single Stage Bootloader

A single stage bootloader consits of a single file that is loaded entirely by the BIOS. This image then performs the steps described above to start the kernel. However, on the x86 you are usually limited to 512 bytes for a first stage (An exception is no-emulation El-Tarito), which is not much. Also, a lot of this size may be dedicated to BIOS structures and FAT headers, which leaves less space to work with.

2.2 Two-Stage Bootloader

A two-stage bootloader actually consists of two bootloaders after each other. The first being small with the sole purpose of loading the second one. The second one can contain all the code needed for loading the kernel. GRUB uses to (or arguably, three) stages.

2.3 Mixed Bootloader

Another way to avoid the 512-bytes barrier is to split the bootloader in two parts, where the first half (512 bytes) can load the rest. This can be achieved by inserting a '512-bytes' break in the ASM code, making sure the rest of the loader is put after the bootsector.

3 Booting Multiple OSes

The easiest way to boot another OS is a mechanism called chainloading. Windows stores something akin to a second-stage bootloader in the boot sector of the partition it was installed in. When installing Linux, writing e.g. LILO or GRUB to the partition boot sector instead of the MBR is also an option. Now, the thing your MBR bootsector can do is to relocate itself (copying from 0x0000:0x7c00 to, traditionally, 0x0060:0x0000), parse the partition table, display some kind of menu and let the user choose which partition to boot from. Then, your (relocated) MBR bootsector would load that partition boot sector to 0x0000:0x7c00, and jump there. The partition boot sector would be none the wiser that there already was a bootsector loaded before, and could actually load yet another bootsector - which is why it's called chainloading.

You see that with displaying a menu in some intelligible way and accepting keystrokes, such a multi-option bootloader can get quite complex rather quickly. We didn't even touch the subject of boot from extended partitions, which would require sequentially reading and parsing multiple extended partition tables before printing the menu.

Taken to the extreme, bootmanagers like that can become as complex as a simple OS, GRUB being a good example: It offers reading from various filesystems, boot Multiboot kernels, chainloading, loading initrd ramdisks etc, etc.