Boot Sequence

From OSDev Wiki

Contents

POST

When a computer is switched on or reset, it runs through a series of diagnostics called POST - **P**ower-**O**n **S**elf-**T**est. Part of this sequence is the scanning for a bootable device.

Old computers would always look for a bootable floppy disk in drive `A:\` first, then for a bootable partition on drive `C:\`. Modern BIOS' allow to configure a scan sequence (e.g., "C, SCSI, A").

Master Boot Record

A device is "bootable" if it carries a boot sector with the byte sequence 0x55, 0xAA in bytes 511 and 512 respectively. When the BIOS finds such a boot sector, it is loaded into memory at a specific location; this is *usually* `0x0000:0x7c00` (segment 0, address 0x7c00). However, some BIOS' load to `0x7c0:0x0000` (segment 0x07c0, offset 0), which resolves to the same physical address, but can be surprising.

When the wrong CS:IP pair is assumed, absolute near jumps will not work properly, and any code like `mov ax,cs; mov ds,ax` will result in unexpected variable locations. A good practice is to enforce CS:IP at the very start of your boot sector.

AsmExample:

```
        ORG 0x7C00
        jmp 0x0000:start
        start:
```

or

```
        ORG 0
        jmp 0x07C0:start
        start:
```

Execution is then transferred to the freshly loaded boot record. On a floppy disk, all 512 bytes of the boot record may contain executable code. (Well, actually 510 bytes due to the two-byte signature.)

> *Is the above information on floppy disk boot records correct?*
>
> > *I second this information. The 'Bios Parameters Block' of a floppy is something that DOS&FAT enforces, but it's perfectly possible to use a floppy without this (except that you have very little clue about what the floppy will be and that you may fail to read a 720KB floppy without this if you cannot tell your OS it's not a 1.44MB disk ...* -- PypeClicker

On a hard drive, the so-called *Master Boot Record* (MBR) holds executable code at offset 0x0000 - 0x01bd, followed by table entries for the four primary partitions, using sixteen bytes per entry (0x01be - 0x01fd), and the two-byte signature (0x01fe - 0x01ff).

The layout of the table entries is as follows:

Offset	Size (bytes)	Description
0x00	1	Boot Indicator (0x80=bootable, 0x00=not bootable)
0x01	1	Starting Head Number
0x02	2	Starting Cylinder Number (10 bits) and Sector (6 bits)
0x04	1	Descriptor (Type of partition/filesystem)
0x05	1	Ending Head Number
0x06	2	Ending Cylinder and Sector numbers
0x08	4	Starting Sector (relative to beginning of disk)
0x0C	4	Number of Sectors in partition

More detailed information about MBR structure and partition types can be found on OSRC (http://www.nondot.org/sabre/os/articles/TheBootProcess/) . The MBR is usually written by disk partitioning programs such as FDisk, Disk Druid, Ranish Partition Manager (http://www.ranish.com/part/) etc.

Early Environment

This early execution environment is highly implementation defined, meaning the implementation of your particular BIOS. *Never* make any assumptions on the contents of registers. They might be initialized to 0, but they might contain a random value just as well.

The CPU is currently in Real Mode, unless you are running on one of those rare BIOS' which believe they're doing you a favor by activating Protected Mode for you. (Which means you not only have to write code for activating protected mode on any *other* hardware, but should also add a test condition if it's already activated.)

Kernel Image

Now we jump two steps ahead and look at where we want to go: Our kernel image. Your boot record would be easiest if it could just copy the kernel image from disk to memory and jump to some given offset. Unfortunately, unless you take extra precautions, your compiler adds all sort of startup code, relocation tables etc. To get a "flat binary" that can be loaded in this simple copy-and-run way, you have to tell GCC:

```
gcc -c my_kernel.c
ld my_kernel.o -o kernel.bin --oformat=binary -Ttext=0x100000
```

The `-c` switch tells GCC to stop right after compilation, i.e. not to link the object file.

The `--oformat=binary` switch tells the linker you want your output file to be a plain binary image (no startup code, no relocations, ...).

The `-Ttext=0x100000` tells the linker you want your "text" (code segment) address to start at the 1mb memory mark. Since you do not link in any relocation tables, the linker has to resolve all references at link time, and has to know where the executable will be loaded to.

You are of course obliged to load your kernel image to the correct offset, or the references the linker did set up will be invalid.

> *It appears that some versions of `ld` prefer `-oformat` rather than `--oformat`. If you're trying to use this tutorial on a non-x86 platform, run `ld --help` if weird things occur.*
>
> *Make also sure that you correctly set the base of* **all** *the sections. Especially when using a linker script, omitting to set the base of `.rodata` or one of the others may make your kernel grew up to 1MB suddenly just because the linker has created a file that can contain both a 0-based `.rodata` and a 1MB-based `.text` ... And if you're trying to run a Higher Half Kernel, it may even crash the linker while it tries to create a 3GB binary file full of zeroes...*
>
> *Additional related problems may occur when using -Ox type parameters to GCC, in this case strings can be moved from `.rodata` to similar named sections. If your linker script doesn't account for this then your image may expand. To prevent this happening, `.rodata*` should be used instead of `.rodata`.*

You may also check the OSD gotchas (http://my.execpc.com/~geezer/osd/gotchas/) to look at well-known issues with common OSdev tools.

Loading

Now we know *what* we have to load, let's see *how* we get it loaded.

If booting from hard drive, you have only 446 bytes available for your boot record. Looking at the list of things to do before your kernel image can run, you will agree that this is not much:

- determine which partition to boot from (either by looking for the active partition, or by presenting the user with a selection of installed operating systems to chose from);
- determine where your kernel image is located on the boot partition (either by interpreting the file system, or by loading the image from a fixed position);
- load the kernel image into memory (requires basic disk I/O);
- enable protected mode;
- preparing the runtime environment for the kernel (e.g. setting up stack space);

You don't have to do things in this order, but all of this has to be done before you can call `kmain()` or whatever you called your kernel entry function.

To make things worse, GCC generates protected mode executables only, so the code for this early environment is one of the Things You Cannot Do With C.

There are several approaches to this problem:

- **Geek loading**: Squeeze everything from the above list into the boot record. This is next to impossible, and does not leave room for any special-case handling or useful error messages.
- **One-stage loading**: Write a stub program for making the switch, and link that in front of your kernel image. Boot record loads kernel image (below the 1mb memory mark, because in real mode that's the upper memory limit!), jumps into the stub, stub makes the switch to Protected Mode and runtime preparations, jumps into kernel proper.
- **Two-stage loading**: Write a *separate* stub program which is loaded below the 1mb memory mark, and does everything from the above list.

The Traditional Way

Traditionally, the MBR relocates itself to `0x0000:0x0600`, determines the active partition from the partition table, loads the first sector of that partition (the "partition boot record") to `0x0000:0x7c00` (hence the previous relocation), and jumps to that address. This is called "chain loading". If you want your self-written boot record to be capable of dual-booting e.g. Windows, it should mimic this behaviour.

Easy Way Out

Unless you really want to be Rolling Your Own Bootloader (record / stubs) for the educational value, we recommend using readily available bootloaders.

The most prominent one is GRUB, a two-stage bootloader that not only provides a boot menu with chainloading capability, but initializes the early environment to a well-defined state (including Protected Mode and reading various interesting information from the BIOS), can load generic executables as kernel images (instead of requiring flat binaries like most other bootloaders), supports optional kernel modules, various file systems, and if `./configure`'d correctly, Diskless Booting.

See Also

Articles

Threads

External Links

- Jun 2008: How Computers Boot Up (http://duartes.org/gustavo/blog/post/how-computers-boot-up) by Gustavo Duarte.
- Jun 2008:The Kernel Boot Process (http://duartes.org/gustavo/blog/post/kernel-boot-process) by Gustavo Duarte.
- IBM developerWorks' Inside the Linux boot process (http://www.ibm.com/developerworks/library/l-linuxboot/index.html) a very good, illustrated overview from BIOS to userspace.