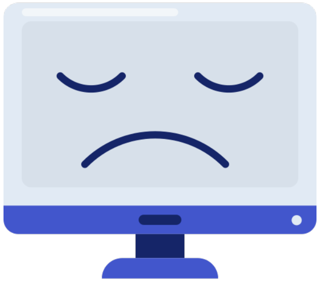
FiascOS documentation



Cyril THOMAS

September 8, 2024

Contents

1 Introduction

This project is intented to be a learning project, not a real application. I wanted to learn how relatively modern computers work in details, and how they differ from simpler microcontrollers.

2 The boot process

When a (relatively) modern computer is turned on, the first thing that has control over it is the BIOS (Basic Input/Output System). It's job is to initialize power control systems, input/output systems, and find a suitable drive to boot. In order to do that, it goes through every disk present on the system, and searches their first sector (the first 512 bytes) for a magic marker (the two bytes **0xAA55** at address **0x1FE**) indicating that the drive is bootable. When one is found, the first sector is loaded in RAM at address **0x7C00**, and the processor jumps to this address, giving the OS control over the processor.

At this stage, the processor is in 16-bit real mode. This means that we have a limited access to the memory, the interruptions, and the connected devices. In this mode, we only have access to 1 MiB of memory, because we can only use 20-bit addresses. In order to use 20-bit addresses with 16-bit registers, there is a system of segments and offsets, usually represented in this way : **SEGMENT:OFFSET**. The real physical address can then be reconstructed using this formula :

$$\text{ADDR} = (\text{SEGMENT} << 4) + \text{OFFSET} = \text{SEGMENT} \times 16 + \text{OFFSET}$$

It also means that in this mode, any address actually has 4096 ways to point to it. For example, addresses **12ab:34cd** and **12ac:34bd** both translate to address **0x15f7d**

The role of this first sector is to initialize the memory layout, load more code, and then switch to 32-bit protected mode.

In real mode, we can use the BIOS to perform some operations for us. For example, reading data from the disk and loading it into memory can be done using the BIOS interrupt 0x13. By putting the required data into the CPU registers and calling this interruption, the BIOS will handle this operation for us. In the case of FiascOS, these interruptions are used when booting to load more code and to print status messages to the screen.

In the case of FiascOS, the memory is initialized according to the following memory map before leaving the bootsector for the kernel :

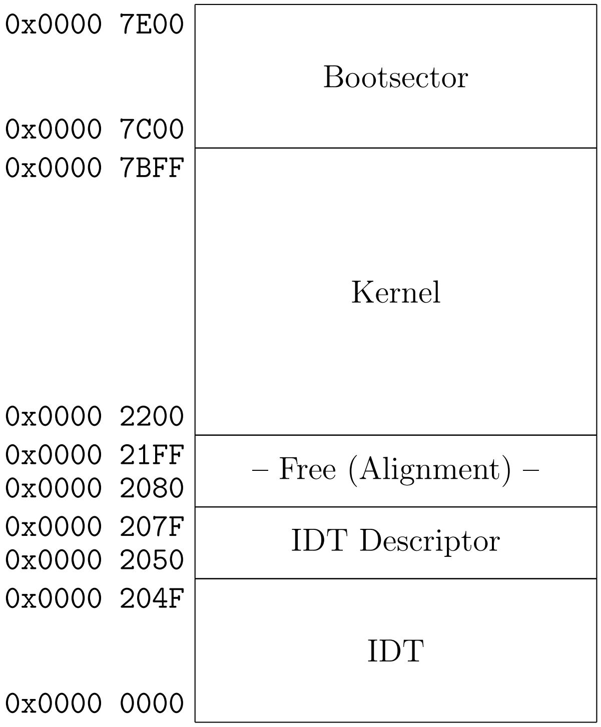


Figure 1: RAM layout when jumping into the kernel

3 Building the image file

3.1 Build process

For the kernel, no relocatable code is used. All pieces of code are written to be loaded at a specific address in RAM. Everything is assembled using NASM to a binary file directly. An include file is included in each .asm code file, so that function addresses can be used in the whole project.

In order for each file to be assembled for the correct address, the **[org 0xXXXX]** directive is used in every piece of non-relocatable code. This indicates to NASM that the code is supposed to be loaded at address **0xXXXX**, and so all memory references need to be adjusted to take this offset into account.

A makefile is used for the build process for convenience. It assembles every file to a binary or elf file as required, and creates the final .bin image using a linker. A linker file is used to store each section of code and data at the right address in the file, so that when the OS tries to load some code, it knows where it is on the disk.

3.2 layout of the file

The layout of the image file is controlled by the **linker.ld** file located in the **Build** directory.
The final image file can be decomposed this way :

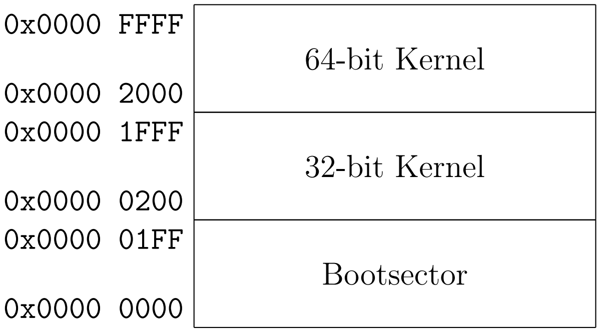


Figure 2: Boot disk layout

4 Functions definition and usage

This section will hold data about the various functions and interrupts that are used in FiascOS.

4.1 16 bit functions

4.1.1 disk_load_16_bits

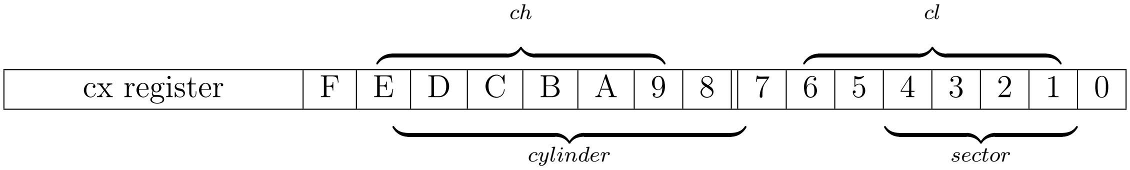
This function is available in the bootsector to load data from the disk to memory using BIOS interrupt 13h. It always reads from Head 0.

<u>**Parameters**</u> :

register	description	range
ah	operation code	0x02
dh	number of sectors to read	0x01 - 0x80
dl	disk number	0x00 - 0xff
ch	cylinder number	0x000 - 0x3ff
cl	number of the first sector to read	0x01 - 0x11
es:bx	address in RAM to load at	0x000 0000 - 0x100 feff

When starting, the BIOS loads the drive number it booted from in dl. This useful when trying to load more code from the boot disk.

The ch and cl parameters may be a bit confusing because the ch parameter can contain values above **0xFF**, that the ch register cannot hold. In reality the cx register contains the cylinder on the 10 most significant bytes and the sector on the 6 least significant bytes.



In case of an error where we couldn't read expected number of sectors, an error message is printed to the screen, and the function enters an infinite loop. In case of a more generic error (indicated by the carry flag being set after calling the interrupt), a generic disk error message is displayed before entering the infinite loop.

Returns :

register or address	description	range
carry flag	error flag	0b0 - 0b1
ah	return code	0x00 - 0xff
al	number of sectors actually read	0x00 - 0x80

Return codes :

code	meaning
0x00	no error
0x01	invalid request
0x02	bad address mark
0x04	sector ID bad or not found
0x05	reset failed
0x08	DMA failure
0x09	DMA overrun: attempted to write across a 64K-byte boundary
0x0A	bad sector flag detected
0x0B	bad cylinder flag detected
0x20	controller failure
0x80	timeout
0xaa	drive not ready
0xbb	undefined error

4.2 32 bit functions

4.2.1 vga_print

The function is available in the 32-bit kernel to print text to the screen is VGA text mode.

Parameters :

register	description	range
ebx	address of string to print	0x0000 0000 - 0xffff ffff
al	Line number to print on	0x00 - 0xff
ch	Color to print the text	0x00 - 0xff
cl	Column number to print the text on	0x00 - 0x4f

Table 1: vga_print inputs

The function prints text strings. The address of the string to print needs to be stored in the **ebx** register, and be null-terminated.

The **al** and **cl** registers can be used to control the position of the text on the screen.

- **al** can take values from **0x00** to **0xff**. Values above 24 (**0x18**) will cause the terminal to be scrolled and the text on higher lines to be erased.
- **cl** can take values from **0x00** to **0x4f**, allowing for 80 characters per line. When this limit is exceeded (either **cl** started with a value greater than **0x4f** or the string was long enough to go past the end of the screen), text will be wrapped around on the line below.

The function can only modify memory in the range of **0xb8000** to **0xb8fa0**. Write attempts to memory regions outside of this range will cause the function to exit.

ch can be used to set the color of the text. Only a single value can be used, swapping colors during the printing of the text is not supported. IT uses the standard IBM 16 color scheme, with the high-order nibble controlling the background and the lower-order nibble controlling the text color.

The following table summarizes the available colors in this mode:

ch register	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0

Table 2: Available colors in VGA text mode

Returns :

This function does not return any value

5 Descriptor Tables layout

5.1 The GDT (Global Descriptor Table)

The GDT is a memory area containing information about the different memory segments defined by the OS.

The GDT is described by a GDT Descriptor structure, that has the following layout for 32-bit mode:

47 16	15 0
Offset (32 bits)	Size (16 bits)

And the following layout when in 64-bit mode :

79	15 0
Offset (64 bits)	Size (16 bits)

Size is equal to the size in bytes of the GDT - 1. This -1 is present because the GDT can have a size of 65536 bytes (8192 entries), but a 16-bit integer can only hold values up to 65535. This poses no problem since the GDT cannot have a size of 0 bytes.

Offset is equal to the linear address of the GDT (not the physical address, paging applies).

For the GDT to be used by the CPU, the GDTR register needs to point to the GDT Descriptor. The value of the GDTR register can be updated using the LGDT instruction :

```
LGDT Address_of_GDT_Descriptor
```

A GDT is mandatory in protected and long modes, but not necessary in real mode.

An entry in the GDT is called a segment descriptor. Entry 0 is not a valid entry, and cannot be used to describe a segment. An entry 8 bytes long and has the following layout in 32-bit mode :

63 56	55 52	51 48	47 40	39 32
Base (31-24)	Flags	Limit (19-16)	Access byte	Base (23-16)

31 16	15 0
Base (15-0)	Limit (15-0)

In 64-bit long mode, a single segment descriptor takes the space of two GDT entries, to make sure that it can hold a 64-bit base address. This is their layout :

127 96
Reserved

95 64
Base (63-32)

63 56	55 52	51 48	47 40	39 32
Base (31-24)	Flags	Limit (19-16)	Access byte	Base (23-16)

31 16	15 0
Base (15-0)	Limit (15-0)

6 Bibliography

References

[1] Base image for the logo : https://www.flaticon.com/free-icons/not-found

[2] Useful tutorial to get started on OS development : https://github.com/cfenollosa/os-tutorial

[3] Lots and lots of resources about OS development on this website : https://wiki.osdev.org/