Our OS

(Bible link to understand - <http://wiki.osdev.org/Main_Page>)

<http://mikeos.sourceforge.net/write-your-own-os.html>

read system initialization : <http://wiki.osdev.org/System_Initialization_%28x86%29>

When a PC is powered-up, it starts executing the **BIOS** (Basic Input/Output System), which is essentially a mini-OS built into the system. It performs a few hardware tests (eg memory checks) and typically spurts out a graphic (eg Dell logo) or diagnostic text to the screen. Then, when it's done, it starts to load your operating system from any media it can find. Many PCs jump to the hard drive and start executing code they find in the **Master Boot Record** (MBR or sector “0”), a 512-byte section at the start of the hard drive; some try to find executable code on a floppy disk (boot sector) or CD-ROM.

see the internal paritioning of MBR <http://linuxgazette.net/156/misc/prestia/mbr.jpg>

On non-[partitioned](http://en.wikipedia.org/wiki/Partition_(computing)) storage devices, VBR is the first sector of the device.

BIOS: It provides an abstraction layer for the hardware, i.e. a consistent way for application programs and operating systems to interact with the keyboard, display, and other input/output devices.

This all depends on the boot order - you can normally specify it in the BIOS options screen. The BIOS loads 512 bytes(bootsector or MBR) from the chosen media into its memory, and begins executing it. This is the bootloader, the small program that then loads the main OS kernel or a larger boot program (eg GRUB/LILO for Linux systems, see image here <http://upload.wikimedia.org/wikipedia/commons/1/12/GRUB_screenshot.png> ). This 512 byte bootloader has two special numbers(sometimes referred as magic numbers) at the end to tell the OS that it's a boot sector - we'll cover that later.(AA55)

size of GRUB is exactly 446 bytes! which fits into MBR’s bootloader partition.

Q)Where does the cpu execute the first instruction from?

Ans. Instruction located at reset vector (the physical memory address FFFF0h on 16-bit x86 processors and FFFFFFF0h on 32-bit and 64-bit x86 processors), usually pointing to the BIOS entry point inside the ROM. This memory location typically contains a jump instruction that transfers execution to the location of the BIOS start-up program.

**Intel software developer’s manual states**

**The top 64kB physical address are mapped to BIOS ROM, not RAM.**

(<http://stackoverflow.com/questions/3274014/0xffff0-and-the-bios?rq=1>)

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf>

<http://stackoverflow.com/questions/5300527/do-normal-x86-or-amd-pcs-run-startup-bios-code-directly-from-rom-or-do-they-cop>

Q) If BIOS chip is ROM, how do we change(and save) the boot order and other settings as seen in?

(http://www.pctipstricks.com/pictures/AwardBIOS-CMOS-Setup-mainscreen.jpg)

Ans. CMOS now comes into picture!! CMOS is [NVRAM](http://en.wikipedia.org/wiki/NVRAM) capacity of 512 [bytes](http://en.wikipedia.org/wiki/Byte), which is generally sufficient for all BIOS settings. Though volatile, it is powered by a small battery when system power was off.

Q) How is BIOS ROM mapped into address space?

<http://stackoverflow.com/questions/7804724/how-is-the-bios-rom-mapped-into-address-space-on-pc?rq=1>

Ans. Real mode vs Protected mode:

**Real Mode**

**Real Mode** is a simplistic 16-bit mode that is present on all x86 processors. Real Mode was the first x86 mode design and is use by many early operating system before the birth of [Protected Mode](http://wiki.osdev.org/Protected_Mode). For compatibility purposes, all x86 processors begin execution in Real Mode.

In [Real Mode](http://wiki.osdev.org/Real_Mode) you use a logical address in the form A:B to address memory. This is translated into a physical address using the equation:

Physical address = (A \* 0x10) + B

The registers in pure real-mode are limited to 16 bits for addressing. 16 bits can represent any integer between 0 and 64k. This means that if we set A to be a fixed value and allow B to change we can address a 64k area of memory. This 64k area is called a segment.

**Protected mode**

In [Protected mode](http://wiki.osdev.org/Protected_mode) you use a logical address in the form A:B to address memory. As in [Real Mode](http://wiki.osdev.org/Real_Mode), A is the segment part and B is the offset within that segment. The registers in protected mode are limited to 32 bits. 32 bits can represent any integer between 0 and 4 GiB.

Because B can be any value between 0 and 4GiB our segments now have a maximum size of 4 GiB (Same reasoning as in real-mode).

Now for the difference.

In protected mode A is not an absolute value for the segment. In protected mode A is a selector. A selector represents an offset into a system table called the [Global Descriptor Table](http://wiki.osdev.org/Global_Descriptor_Table) (GDT). The GDT contains a list of descriptors. Each of these descriptors contains information that describes the characteristics of a segment.

Given a logical address A:B (Remember that A is a selector) we can determine the physical address it translates to using:

Physical address = Segment Base (Found from the descriptor GDT[A]) + B

Below you'll find a list of cons and pros. These are mostly 'in comparison to [Protected Mode](http://wiki.osdev.org/Protected_Mode)'.

### Cons

* Less than 1 MB of RAM is available for use.
* There is no hardware-based memory protection (GDT), nor virtual memory.
* There is no built in security mechanisms to protect against buggy or malicious applications.
* The default CPU operand length is only 16 bits.
* The memory addressing modes provided are more restrictive than other CPU modes.
* Accessing more than 64k requires the use of segment register that are difficult to work with.

### Pros

* The [BIOS](http://wiki.osdev.org/BIOS) installs device drivers to control devices and handle interrupt.
* [**BIOS functions**](http://wiki.osdev.org/BIOS#BIOS_functions) **provide operating systems with a advanced collection of low level API functions.**
* Memory access is faster due to the lack of descriptor tables to check and smaller registers.

A CPU that is initialized by the [BIOS](http://wiki.osdev.org/BIOS) starts in [Real Mode](http://wiki.osdev.org/Real_Mode). Enabling Protected Mode unleashes the real power of your CPU. However, it will prevent you from using most of the BIOS interrupts, since these work in Real Mode(unless you have virtual mode). In [Protected mode](http://wiki.osdev.org/Protected_mode), almost all BIOS functions become unavailable, and trying to call them nonetheless will result in exceptions or unreliable responses (because of the different way **segment** values are handled)

Whether the CPU is in [Real Mode](http://wiki.osdev.org/Real_Mode) or in Protected Mode is defined by the lowest bit of the CR0 (control register).

Protected mode was introduced in 80286

If time permits learn about-

EFI , FAT , BPB bios parameter block , OEM

For higher resolution-

<http://superuser.com/questions/357328/how-computers-display-raw-low-level-text-and-graphics>

VBE(VESA BIOS Extensions)

<http://en.wikipedia.org/wiki/VESA_BIOS_Extensions>

Putpixel in VESA

<http://www.oocities.org/siliconvalley/pines/8864/vesa.txt>

Mode 13h 256colors

<http://en.wikipedia.org/wiki/Mode_13h>

<http://www.asmcommunity.net/forums/topic/?id=30006>

Putpixel in mode 13h direct mem access faster than 10h bios interrupt calls!

<https://courses.engr.illinois.edu/ece390/books/labmanual/graphics-mode13h.html>

Putpixel in VESA VBE direct mem access faster than 10h bios interrupt calls!

<https://groups.google.com/forum/#!topic/comp.lang.asm.x86/Z8nVCgBdLmA>

Bootloader to kernel (more than 512bytes)

<http://www.codeproject.com/Articles/737545/Writing-a-bit-dummy-kernel-in-C-Cplusplus>

**KERNEL**

In [computing](http://en.wikipedia.org/wiki/Computing), the kernel is a [computer program](http://en.wikipedia.org/wiki/Computer_program) that manages [input/output](http://en.wikipedia.org/wiki/Input/output) requests from [software](http://en.wikipedia.org/wiki/Software), and translates them into [data processing](http://en.wikipedia.org/wiki/Data_processing) instructions for the [central processing unit](http://en.wikipedia.org/wiki/Central_processing_unit) and other [electronic components](http://en.wikipedia.org/wiki/Electronic_component) of a [computer](http://en.wikipedia.org/wiki/Computer). The kernel is a fundamental part of a modern computer's [operating system](http://en.wikipedia.org/wiki/Operating_system).

When a computer program (in this context called a [*process*](http://en.wikipedia.org/wiki/Process_(computing))) makes requests of the kernel, the request is called a[system call](http://en.wikipedia.org/wiki/System_call).

Q)What is a flat binary program?

Ans.

A flat binary program can basically be defined as a program where the **entry point routine**(such as main() ) is always at the first byte of the program file.

Q)Why do we output a flat binary using NASM?(where )

Ans.

The BIOS does not at all know what internal format this program file is or what it does. Because of this, it treats the Boot Loader as a **flat binary program**. It loads whatever is on the **Boot Sector** of the **Boot Disk** and "jumps" to the first byte of that program file.

Q)How to see the flat binary?

Ans.

**hexdump -C *file-path***

***200 in hexadecimal is 512. That explains 512bytes sector***

Q)Why haven’t we used any segments/sections in nasm?

Ans.

Nasm's sections/segments in the binary format are practically useless, you can have them in the source file, but there's barely a trace of them in the output binary. If you assemble the source file into some other output format, e.g. obj, coff/aout, elf, win32 then the sections/segments will make it into the output file and the linker will be able to do some meaningful work with them (create a proper executable file for the OS with pieces of it marked as code (executable), data (non-executable), etc). Only in the obj format sections/segments represent x86 segments.

System programming vs Application programming- terms

Q)Why did we choose NASM over TASM,MASM?

Ans.

The Netwide Assembler (NASM) can generate flat binary 16bit programs, while most other assemblers (Turbo Assembler (TASM), Microsoft's Macro Assembler (MASM)) cannot.

We use VirtualFloppyDrive to create the floppy image.

**Read mode segment:offset problems!!**

Technically, there is exactally **4,096** different combinations of segment:offset that can refer to the same byte in memory -- This is for each byte in memory!

By changing the segment and offset values, you can find different segment:offset pairs will yeld the same absolute address. Why? Because they both refer to the same memory location!

For example, all of the below addresses refer to our bootloader at 0x7C00:

0007:7B90 0008:7B80 0009:7B70 000A:7B60 000B:7B50 000C:7B40

Q)Why real mode is said to offer no memory protection?

Ans.

As see above,in Real Mode, you can access every byte in memory, over 4,000 different ways, being able to overlap segments that could potentally corrupt that area of memory without you knowing. This is what is ment by Real Mode not having any **Memory Protection**.

OEM parameter block-?

Floppy disks have

* 18 sectors per track
* 63 total tracks
* 2 heads

volume boot record

design of fat system

<http://www.eit.lth.se/fileadmin/eit/courses/eitn50/Projekt1/FAT12Description.pdf>

file system

data cluster

file allocation table

<http://wiki.osdev.org/FAT>

FAT12 code http://board.flatassembler.net/topic.php?p=159541

**FILE SYSTEM**

to be learnt

Commands while building-

<http://en.wikipedia.org/wiki/Dd_(Unix)>

1. Create floppy image

dd if=/dev/zero bs=512(512 bytes per sector or block size) count=2880(total no of sectors) > floppy.img

why did we choose disk as zero?

<http://en.wikipedia.org/wiki//dev/zero>

**/dev/zero** is a [special file](http://en.wikipedia.org/wiki/Special_file) in [Unix-like](http://en.wikipedia.org/wiki/Unix-like) [operating systems](http://en.wikipedia.org/wiki/Operating_system) that provides as many [null characters](http://en.wikipedia.org/wiki/Null_character) ([ASCII](http://en.wikipedia.org/wiki/ASCII) NUL, 0x00) as are read from it.[[1]](http://en.wikipedia.org/wiki//dev/zero#cite_note-1) One of the typical uses is to provide a character stream for initializing data storage.[[2]](http://en.wikipedia.org/wiki//dev/zero#cite_note-2)

2) Copy our bootloader to bootsector of our floppy image

dd conv=notrunc if=bootload.bin of=floppy.img

The notrunc conversion option means do not truncate the output file — that is, if the output file already exists, just replace the specified bytes and leave the rest of the output file alone. Without this option, dd would create an output file 512 bytes long. Whereas we want the original floppy size to be maintained to 1.44MB

Each of the "Records in" and "Records out" lines shows the number of complete blocks transferred + the number of partial blocks,

Power Management

<http://wiki.osdev.org/Shutdown>

APM vs ACPI

How does a computer restart?

<http://superuser.com/questions/294681/how-does-a-computer-restart-itself>

See how we use the CPU reset line using 8042 keyboard controller

<http://wiki.osdev.org/%228042%22_PS/2_Controller>

Q)Why doesn’t our OS boot on a Intel Mac?

Ans.

The following applies not just to USB hard disks, but to any storage device that is not considered "internal". That includes USB flash drives, SD cards and other memory cards, as well as hard drives attached through Firewire or other connections.

Booting Windows or Linux from an external disk is not well-supported by Apple’s firmware.

Apple’s [Boot Camp FAQ](http://support.apple.com/kb/HT3777) states that you can only install Windows on internal hard drives.

**Boot device number**

<http://stackoverflow.com/questions/11174399/pc-boot-dl-register-and-drive-number>

STRUCTURE OF A FAT12 FORMATTED DISK

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot Sector | Extra Reserved Sectors | File Allocation Table 1 | File Allocation Table 2 | Root Directory (FAT12/FAT16 Only) | Data Region containng files and directories. |

Boot sector, FAT12, bootloader explained

<http://www.brokenthorn.com/Resources/OSDev5.html>

FAT 2 is used for recovery purposes.

Q) How do we get the first sector to Root Directory?

**The Root Directory is right after all of the FATs**. This means...

**if we add up the number of sectors per FAT(9 for our floppy disk), and the reserved sectors(0 for us), we can get the first sector to the Root Directory.** By searching the Root Directory for a simple string (our filename), we can effectivly find the exact sector of the file on disk :)

Root directory

The root directory is a table of 32 byte values that represent information reguarding file and directories. This 32 byte value uses the format:

* **Bytes 0-7 : DOS File name (Padded with spaces)**
* **Bytes 8-10 : DOS File extension (Padded with spaces)**
* **Bytes 11 :** File attributes. This is a bit pattern:
  + **Bit 0 :** Read Only
  + **Bit 1 :** Hidden
  + **Bit 2 :** System
  + **Bit 3 :** Volume Label
  + **Bit 4 :** This is a subdirectory
  + **Bit 5 :** Archive
  + **Bit 6 :** Device (Internal use)
  + **Bit 6 :** Unused
* **Bytes 12 :** Unused
* **Bytes 13 :** Create time in ms
* **Bytes 14-15 :** Created time, using the following format:
  + **Bit 0-4 :** Seconds (0-29)
  + **Bit 5-10 :** Minutes (0-59)
  + **Bit 11-15 :** Hours (0-23)
* **Bytes 16-17 :** Created year in the following format:
  + **Bit 0-4 :** Year (0=1980; 127=2107
  + **Bit 5-8 :** Month (1=January; 12=December)
  + **Bit 9-15 :** Hours (0-23)
* **Bytes 18-19 :** Last access date (Uses same format as above)
* **Bytes 20-21 :** EA Index (Used in OS/2 and NT, dont worry about it)
* **Bytes 22-23 :** Last Modified time (See byte 14-15 for format)
* **Bytes 24-25 :** Last modified date (See bytes 16-17 for format)
* **Bytes 26-27 :** **First Cluster //logical cluster!!**
* **Bytes 28-32 :** **File Size**

FAT12

Each entry is a 12 bit value that represents a cluster. **The FAT is a linked list-like structure with these entrys that helps identify what clusters are in use**.

* **Value marks free cluster :** 0x00
* **Value marks Reserved cluster :** 0x01
* **This cluster is in use--the value represents next cluster :** 0x002 through 0xFEF
* **Reserved values :** 0xFF0 through 0xFF6
* **Value marks bad clusters :** 0xFF7
* **Value marks this cluster as the last in the file :** 0xFF8 through 0xFFF

**The maximum number of files possible is 4,077**

How?

cluster number=12bits=2^12=4096 but out of those, only 002toFEF are used so only 4077 files.

**Root directory and FAT connection!**

A FAT is just an array of these values--thats all. **When we find the starting sector form the Root Directory(by the calculation above), we can look through the FAT to find which clusters to load. How? We simply check the value(bytes 26-27 contain first cluster). If the value is between 0x02 and 0xfef, this value represents the next cluster to load for the file.**

In our case, each cluster is 1 sector. When we get the first sector of Stage 2 (We get this from the root directory by above calculation), we use this sector as the starting cluster number in the FAT. Once we find the starting cluster, we just refrence the FAT to determin the cluster (The FAT is just an array of 12 bit numbers. We just compare this number with the list above to determin what to do with it.)

Excellent pdf on FAT12

<http://www.eit.lth.se/fileadmin/eit/courses/eitn50/Projekt1/FAT12Description.pdf>

**Translation from logical sector number(contained in FAT) to physical sector number**

Translation from physical to logical data sector numbers:

The FAT works off logical data sector values. For the FAT12 system, while determining the logical

sector number from the physical sector number, the following two factors need to be taken into

account.

• From the organization of the disk, it is seen that the first 33 sectors are predefined.

(0-boot sector,1-9 fat1,10-19 fat 2 + 14 sectors for root directory) so 0-32 sectors=33 sectors

The actual data sector that holds user data does not exist in these first 33 sectors and starts at sector number 33 (remember we start with 0).

• The entries in positions 0 and 1 of the FAT are reserved. Therefore, it is entry 2 of the FAT

that actually contains the description for physical sector number 33.

Therefore, physical sector number = 33 + FAT entry number - 2

For example, entry 5 of the FAT would actually refer to physical data sector number 36.

Cold boot (hard reboot) vs Warm boot (soft reboot)

Int 19h Bootstrap loader service http://www.ousob.com/ng/asm/ng229bf.php

This interrupt attempts to load the sector at head 0, track 0, sector 1, on the first diskette into memory at 0:7C00h. If unable, it then attempts to load the sector at head 0, track 0, sector 1 of the first hard disk.

If INT 19h is successful, control is transferred to the first byte of the sector, which has been read in at memory location 0:7C00h. That is, CS is set to 0 and IP is set to 7C00h.

This interrupt is not a substitute for a Ctrl-Alt- Del warm boot. This interrupt is the last action performed by the Power-On Self Test (POST), which is activated by Ctrl-Alt-Del. If a user program attempts to execute this interrupt directly, the machine may lock up.

To correctly reboot within an application program, jump to FFFF:0h in memory. That is, set CS to FFFF and IP to 0. This is the same process that occurs when the machine is first powered on. Following these steps will perform a "cold" boot. To perform a "warm" boot using this method, first set the word at location 0:472h to 1234h. This will simulate the pressing of the Ctrl-Alt-Del.

A cold boot initializes all hardware, tests all hardware, tests RAM, then calls INT 19h to load the bootstrap loader. This process is performed when the computer is turned on or a hardware reset button is pressed.

A warm boot initializes and tests all hardware but does not test RAM. It then calls INT 19h to load the bootstrap loader. This process is performed when Ctrl-Alt-Del is typed.

What is the above ctrl-alt-del being talked about?

Read-

<http://en.wikipedia.org/wiki/Control-Alt-Delete>

It has different functions on different platforms.

On BIOS Perform a [soft reboot](http://en.wikipedia.org/wiki/Soft_reboot) without memory initialization

in pre-boot environment (before an [operating system](http://en.wikipedia.org/wiki/Operating_system) starts)[[4]](http://en.wikipedia.org/wiki/Control-Alt-Delete#cite_note-IBM_1983_PC-TR-4)[[5]](http://en.wikipedia.org/wiki/Control-Alt-Delete#cite_note-IBM_1984_AT-TR-5)[[6]](http://en.wikipedia.org/wiki/Control-Alt-Delete#cite_note-Phoenix_1989_BIOS-6) or in [DOS](http://en.wikipedia.org/wiki/DOS), [Windows 3.0](http://en.wikipedia.org/wiki/Windows_3.0) and earlier versions of [Windows](http://en.wikipedia.org/wiki/Windows) or [OS/2](http://en.wikipedia.org/wiki/OS/2), the key combination [reboots](http://en.wikipedia.org/wiki/Reboot_(computing)) the computer. Starting with [Windows 3.1](http://en.wikipedia.org/wiki/Windows_3.1), the command invokes a [task manager](http://en.wikipedia.org/wiki/Task_manager) or security related component that facilitates ending a Windows session.

Going back in history for the three finger salute ( ctrl-alt-del)

The [soft reboot](http://en.wikipedia.org/wiki/Soft_reboot) function via keyboard was originally designed by [David Bradley](http://en.wikipedia.org/wiki/David_Bradley_(engineer)).[[7]](http://en.wikipedia.org/wiki/Control-Alt-Delete#cite_note-Hughes_2013_History-7) Bradley, as the chief engineer of the [IBM PC](http://en.wikipedia.org/wiki/IBM_Personal_Computer) projec

We used three finger salute to log on right ?

**During a question and answer presentation on 21 September 2013, Gates said "it was a mistake", referring to the decision to use Ctrl+Alt+Del as the keyboard combination to log in to Windows.**

Finally understanding FAT12 !!! (an entry inFAT is cluster number of 12bits..but access is obviously done in 8bits or 16bits!! )

Lets imagine a FAT

starting at some address having bytes as (remember byte=8bits,word=16bits,cluster=12bits)

01011101 0111010 01110101 00111101 0011101 0111010

| | | |

0th cluster 1st cluster 2nd cluster

;Notice all even clusters occupy all of the first byte, but part of the second. Also notice that all odd clusters occupy a part of their first byte, but all of the second!

;Okay, so what we need to do is to read a 2byte (word) value from the FAT

;If the word is even, Mask out the top 4 bits, as it belongs to the next cluster.

;If it is odd, shift it down 4 bits (to discard the bits used by the first cluster.

So suppose our cluster no=2

;our next cluster will be=fat[cluster]=fat[2]=1100 10111100(higher to lower bits!!)

;to get to that we must load, 3rd,4th byte(only lower 4bits) !

;VERY IMPORTANT

;so word to be loaded=cluster+cluster/2=cluster\*3/2

;take example for current cluster=4, you need 6th and 7th bytes

;so word=4+4/2=6 followed by masking!

So algorithm to load a cluster is…(indexing a Fat table is)

1) find byte where current cluster is located using formula

byte= cluster+cluster/2

2) next index this byte into the table. baseadd of fat table + this byte

3) read a word at this index

4) mask bits accordingly if index is even or odd..

5) hence we get the next cluster

6) if this cluster ==0ff8 then it marks end of file!

Another algorithm to convert between a Logical Block Addressing and Cylinder Head Sector addressing.

Since to read sector to memory we call int13h/ah=02h we need to specify no of sectors to be read and from which sector to start!

Working with LBA is easy but to read the sector we need to know where this logical sector lies ( which head, which cylinder ,which physical sector)

AH = 02h

AL = number of sectors to read (must be nonzero)

CH = low eight bits of cylinder number

CL = sector number 1-63 (bits 0-5) high two bits of cylinder (bits 6-7, hard disk only)

DH = head number

DL = drive number (bit 7 set for hard disk)

ES:BX -> data buffer

es:bx points to buffer(buffer located in memory) that holds the bytes which are read from the disk!

;absolute sector = (LBA % sectors per track) + 1

;absolute head = (LBA / sectors per track) % number of heads

;absolute track = LBA / (sectors per track \* number of heads)

Cylinder is nothing but a cylinder of all tracks considering all platter together.

Root directory uses CHS scheme to store the starting cluster number!

To convert from CHS to LBA

From the organization of the disk, it is seen that the first 33 sectors are predefined.

;(0-boot sector,1-9 fat1,10-19 fat 2 + 14 sectors for root directory) so 0-32 sectors=33 sectors

;The actual data sector that holds user data does not exist in these first 33 sectors and starts at sector number 33 (remember we start with 0).

;The entries in positions 0 and 1 of the FAT are reserved. Therefore, it is entry 2 of the FAT

;that actually contains the description for physical sector number 33.

;Therefore, physical sector number = 33 + FAT entry number - 2

;For example, entry 5 of the FAT would actually refer to physical data sector number 36.

; Now we must load the FAT from the disk. Here's how we find out where it starts:

; FAT cluster 0 = media descriptor = 0F0h

; FAT cluster 1 = filler cluster = 0FFh

; Cluster start = ((cluster number) - 2) \* SectorsPerCluster + (start of user)

; = (cluster number) + 31

**BOOTLOADER.asm code explained! (this is loaded into sector 0 of our floppy)**

1) BIOS parameter block

defines what file system is our disk formatted. And the entire description

viz bytespersector,sectorsperclustor,no of fats, etc.

remember this is not code! so we must skip over this BPB

2) The bootstrap loader loads us at 0000:07c0

So our code is executing with cs=0 and ip=07c0 and this is where our bootloader.asm code is being stored.

We setup stack space( as we use pop and push instructions ) as well as buffer space (this is where the files are loaded into memory. buffer is simply a pointer to some memory in our ram.)

we take 8k size buffer and 4k size stack.

remember push decrements stack pointer and pop increments stack pointer.

few calculations are hence done to allocate the above stated space.

1paragraph=16bytes

3) now keep in mind the typical fat12 formatted disk

boot sector(where bootload.asm is present) followed by 2 fat tables (other one for recovery) followed by root directory followed by our files.

Now we read our file from the root directory(it holds 32byte entry for each file present on the disk with 26-27 byte giving first cluster number in chs scheme.)

we know root directory will be in logical sector 19.

how? 1(boot sector) + 2 fats ( 9 sectors each for fat12 system) =19

root directory is 14sectors!

so we call interrupt 13h with appropriate parameters in our registers! and read the whole 14 sectors=14\*512=7168 bytes into buffer (our buffer was 8k)

buffer is pointer by es:bx

Now we have entire directory in our buffer.

To search for file name 0-10 bytes in a 32byte root dir entry contains file name.

We string match our kernel file name with first 11 bytes of an entry.

And then increment by 32bytes the base address to point to the next entry in the root dir.!!!

4) once we got the entry containing the kernel file. we go to its 26th 27th byte.

the root dir stores cluster number in chs scheme. hence we translate it to lba scheme.

5) next we load the FAT into our buffer. fat logical cluster=1

convert this lba to chs and call int13h loadin 9 sectors!

6) now our buffer has entire fat and we have the starting cluster number!

index the cluster number into fat ( how ? cluster=12bits , fat entries are 12bit but we can access either 8 or 16bits at a time! )

so we observe (given above)

and get the formula

word to be loaded= cluster + cluster/2

we load the word indexed into cluster..by masking off the bits accordingly!

And hence get the next cluster number.

Convert this lba cluster into chs! And call int13h and load into our pointer (this is also a buffer pointed to location in memory where our kernel.asm has been fetched from the disk!!! ) cluster=sector for fat12..hence after loading of one cluster ie 512 bytes of kernel.asm

we increment our pointer by 512bytes! so as to fetch the next cluster and avoid overlapping overwriting!

loop this until we get an ending cluster number= 0ff8h

Phew..