Bootloader to boot the first file from a FAT32 file system

What my bootloader can do:

It can load the first file(which should be a file less than 4096 bytes) in a FAT32 file system to the memory and hand over the execution to that program.

Steps to use my Bootloader:

1. Compile the bootloader.

```
nasm <path of the bootloader> -f bin -o boot.bin
```

2. Unmount the USB device.

```
sudo umount <absolute path of the device>
```

3. Format the USB device to FAT32 file system.

```
Sudo /sbin/mkdosfs -F 32 <absolute path of the device>
```

4. Copy the Bootloader to the USB device.

```
Sudo /sbin/mkdosfs -F 32 <absolute path of the device>
```

5. Copy the compiled kernel to the USB device(just as you copy any usual file to the USB device).
6. Restart the computer and boot from the USB device!.

Introduction to the FAT32 file system:

The major difference between FAT12 and FAT 32 is that there is no specific Root directory in FAT32(There can be lots of other differences, but this is the vital difference that needs to be known to do this practical correctly).

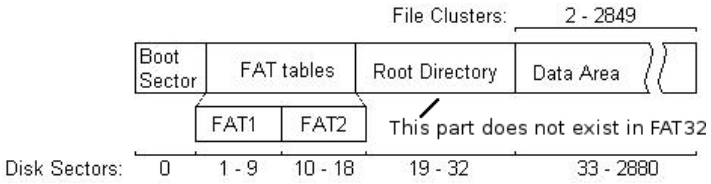

Fig 1: FAT12 file system

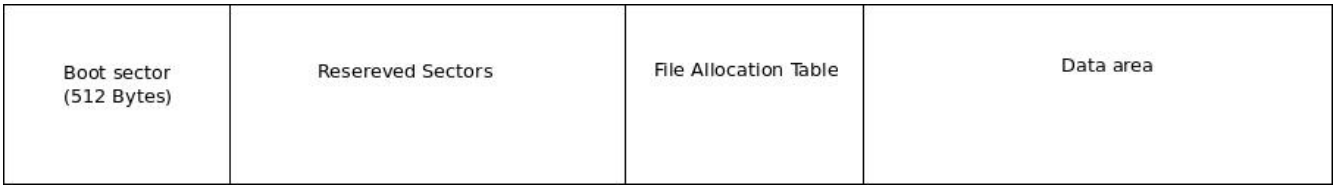

Fig2: FAT32 file system

The length of the Reserved Sectors can vary depending on the USB drive.

To read a file from a FAT32 drive, we should first find out the beginning of the Data area. Then we should find the location of the first directory entry. In that directory entry the location of the file is stored in an offset of 0x1A. By jumping to the location that is indicated by 0x1A offset,we can go to the beginning of the kernel file.

Offset	Length in bytes	Meaning
0x00	8	File name
0x08	3	Extended name
0x0B	1	Atribute byte
0x0C	10	Reserved
0x16	2	Create time
0x18	2	File date
0x1A	2	First cluster number
0x1C	4	File length

Table 1: Structure of a directory entry

For the sake of understanding, lets look at a small example on how to traverse in the FAT32 file system.

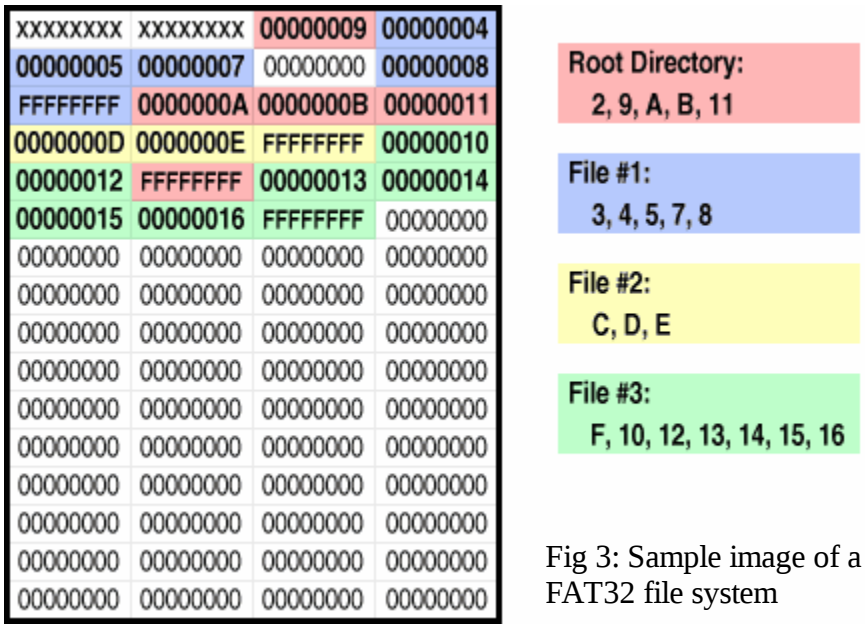

Fig 3: Sample image of a Data area in a FAT32 file system

Note: All the information in an entry,except the information about the next entry ,has been excluded in this example.

The locations of the Directory entries are shown in rose colour. As we can see, the first directory entry points to 0x09 where the next directory entry is can be found. Also if we look at file number 1, its first entry points to 0x04, where the its next entry exists. Every chain has ended in 0xFFFFFFFF which symbolizes the end of a chain.

Explanation of the Bootloader source:

Let's take a look at the code piece by piece.

```
OEM_ID              db      "QUASI-OS"
BytesPerSector      dw      0x0200
SectorsPerCluster   db      0x08
ReservedSectors     dw      0x0020
TotalFATs           db      0x02
MaxRootEntries      dw      0x0000
NumberOfSectors     dw      0x0000
MediaDescriptor     db      0xF8
SectorsPerFAT       dw      0x0000
SectorsPerTrack     dw      0x003D
SectorsPerHead      dw      0x0002
HiddenSectors       dd      0x00000000
TotalSectors        dd      0xFE3B1F00
BigSectorsPerFAT    dd      0x00000778
Flags               dw      0x0000
FSVersion           dw      0x0000
RootDirectoryStart  dd      0x00000002
FSInfoSector        dw      0x0001
BackupBootSector    dw      0x0006
DriveNumber         db      0x00
Signature           db      0x29
VolumeID            dd      0xFFFFFFFF
VolumeLabel         db      "QUASI  BOOT"
SystemID            db      "FAT32   "
```

To read the files from the file system correctly, we need to find out some values of the file system. This info can be simply found out by doing a hex dump of the first 512 Bytes and reading the relevant offsets using a hex reader. The relevant offsets and their uses and lengths are shown below.

Offset in Boot Sector	Length in Bytes	Mnemonic
0x03	8	OEM_Identifier
0x0B	2	BytesPerSector
0x0D	1	SectorsPerCluster
0x0E	2	ReservedSectors

0x10	1	NumberOfFATs
0x11	2	RootEntries
0x13	2	NumberOfSectors
0x15	1	MediaDescriptor
0x16	2	SectorsPerFAT
0x18	2	SectorsPerHead
0x1A	2	HeadsPerCylinder
0x1C	4	HiddenSectors
0x20	4	BigNumberOfSectors
0x24	4	BigSectorsPerFAT
0x28	2	ExtFlags
0x2A	2	FSVersion
0x2C	4	RootDirectoryStart
0x30	2	FSInfoSector
0x32	2	BackupBootSector
0x34	12d	Reserved

Table 2: Boot record layout

Lets see what is the use of each field.

OEM_Identifier:

It is an eight-byte ASCII string that identifies the system that formatted the disk. This has got no particular use.

BytesPerSector:

Indicates the length of a sector in Bytes. Usually this is 512 for FAT32.

SectorsPerCluster:

Indicates how many sectors are in one logical cluster. Allowed values are powers of two from 1 to 128.

ReservedSectors:

Indicates the length of the reserved sectors. For FAT12 and FAT16 this value is usually 1. For FAT32 it is 20h. At least one sector must always be reserved.

NumberOfFATs:

Indicates the number of File Allocation Tables. This value is usually two. FATs are consecutive on the disk: the second copy of FAT goes right after the first copy. At least one copy of FAT should be present.

RootEntries:

This contains the number of the entries in the root directory if root directory is

fixed. It is zero if the root directory is not fixed.As FAT32 file system's root directory can be arbitrarily long, FAT32 disks should contain zero in this field . Otherwise, this field usually contains 512.

NumberOfSectors:

Represents the total number of sectors on the disk. If the number of sectors is greater than 65535, then this field is set to zero and the value will be represented by TotalSectors.

MediaDescriptor:

Indicates what type of a media this is.

Value	Meaning
FF	5 1/4 floppy, 320KB
FE	5 1/4 floppy, 160KB
FD	5 1/4 floppy, 360KB
FC	5 1/4 floppy, 180KB
F9	5 1/4 floppy, 1.2MB
F8	Any Hard Drive
F0	3 1/2 floppy, 1.44MB

Table 3: Media descriptor information

SectorsPerFAT:

This contains the number of sectors in one FAT. This field is zero for FAT32 drives and BigSectorsPerFAT contains the actual value.

SectorsPerTrack:

Indicate the number of sectors for a single track.

SectorsPerHead:

It is the number of sectors grouped under one head.

HiddenSectors:

Indicates the number of sectors between the beginning of this partition and the partition table.

TotalSectors:

This contains the number of total sectors if it is greater than 65535.

BigSectorsPerFAT:

As the drive's file system is FAT 32, this entry includes Number of sectors in one FAT.

Flags:

This is defined only for FAT32 disks. They are defined differently for FAT12 and FAT16. If the left-most bit of ExtFlags value is set then only the active copy of FAT is changed. If the bit is cleared then FATs will be kept in synchronization. Disk analyzing programs should set this bit

only if some copies of the FAT contain defective sectors. Low four bits define which copy should be active.

FSVersion :

Indicates the version of the file system.

RootDirectoryStart :

contains the number of the first cluster for the root directory. As you can see, finally the root directory became stored like any other directory, in the cluster chain. This also implies that it may grow as needed.

FSInfoSector:

This is the sector number for the file system information sector.

BackupBootSector:

This is the sector number for the backup copy of the boot sector. This copy can be used if the main copy was corrupted.

DriveNumber:

Indicates the Physical Drive Number.

Signature:

Extended boot signature.

VolumeID:

Acts as the serial number for the drive.

Volume Label:

Give's a label(name) to the drive.

SystemID:

Indiactes FAT file system type.

```
mov     al, BYTE [TotalFATs]
mul     WORD[BigSectorsPerFAT]
add     ax, WORD [ReservedSectors]
mov     WORD [datasector], ax
```

Length of the FAT(in sectors)= TotalFATs * BigSectorsPerFAT

Position of the beginning of Data sectors= Length of the FAT + ReservedSectors

So, using the above code segment we can find the sector number of the beginning of the data sector.


```
mov     ax, WORD[RootDirectoryStart]
call ClusterLBA
mov     bx, 0x0200
call    ReadSectors
```

Using this code code, the first file entry is loaded to the offset 0x0200 in segment 0x7C00.

```
mov     di, 0x0200 + 0x20
```

The first 32 bytes of the Data area contains some other information(This is shown in figure 3 also). So we need to skip that data to get to the first file entry. This code segment does that task.

```
mov     dx, WORD [di + 0x001A]
```

As said earlier, the actual location of the file in the hard disk is stored in the directory entry in an offset of 0x001A. The above code segment points the dx register to that offset.

```
read:
        mov ax, 0100h
        mov es, ax
        mov bx, 0

        mov cx, 0x0008
        mov ax, WORD[cluster]
        call ClusterLBA
        call ReadSectors
```

The es register is set to 0x0100 and bx register is set to 0x0000 so that the kernel will be loaded to memory segment 0x0100 in a offset of 0x0000. Cx register is set to 0x0008 so that the whole cluster is read in to the memory.Then the kernel image is read in to memory using **ClusterLBA** and **ReadSectors** functions.

Note: As the code has been written only to load the first cluster to the memory (which is 4096 bytes in length), only kernels less than 4096 bytes can be booted using the bootloader.

```
push    WORD 0x0100
push    WORD 0x0000
retf
```

Now the kernel has been loaded to the memory segment 0x0100 starting in the offset 0x0000. Above command makes the program flow to jump to that memory segment and offset and execute the kernel.

Brief introduction to the functions:

Let's take a brief look at the functions that have been used in this code. I did not change any of these functions, they are the same functions that were used in the FAT12 bootloader.

ReadSectors:

The functionality of this function is simply to read the hard disk using interrupt 13 till a file record is found. To do so the registers should be set as follows.

AH	02h
AL	Sectors To Read Count
CH	Track
CL	Sector
DH	Head
DL	Drive
ES:BX	Buffer Address Pointer

After interrupt 13 is called the registers will be set as follows.

CF	Set On Error, Clear If No Error
AH	Return Code
AL	Actual Sectors Read Count

ClusterLBA:

This function simply calculates the Logical Block Address for a given cluster number which is stored in the ax register.

LBACHS:

This function is used calculates the Sector, Head, Track numbers from the given LBA address.

References:

Information on FAT32 file system:

http://www.pjrc.com/tech/8051/ide/fat32.html

http://support.microsoft.com/kb/q140418/

http://home.freeuk.net/foxy2k/disk/disk2.htm

http://www.easeus.com/data-recovery-ebook/root-directory-management-in-FAT32.htm

http://home.teleport.com/~brainy/fat32.htm

http://averstak.tripod.com/fatdox/bootsec.htm

http://www.easeus.com/resource/fat32-disk-structure.htm

http://skola.jozjan.net/Operacne%20systemy/OS/fat32.htm

http://en.wikipedia.org/wiki/File_Allocation_Table

Guides on Bootloaders:

http://www.viralpatel.net/taj/tutorial/hello_world_bootloader.php

http://www.brokenthorn.com/Resources/OSDevIndex.html

http://www.daniweb.com/forums/thread316156.html

Information about Registers:

http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

Assembly guides:

http://cyberasylum.wordpress.com/2010/11/19/assembly-tips-and-tricks/

http://en.wikipedia.org/wiki/Assembly_language

http://www.cs.uaf.edu/2006/fall/cs301/support/x86/

Other resources:

http://en.wikipedia.org/wiki/INT_13#INT_13h_AH.3D02h:_Read_Sectors_From_Drive

http://en.wikipedia.org/wiki/Logical_block_addressing