FAT (File Allocation Table) File System Tutorial

http://students.cs.byu.edu/~cs345ta/lectures/seamons/FAT.pdf

1

FAT File System

- Originated in the late 1970s and early 1980s
- The file system supported by MS-DOS
- Originally developed as a simple file system suitable for floppy disk drives less than 500K in size
- Currently there are three FAT file system types: FAT12, FAT16, FAT32
  - FATnn – nn is the number of bits in each entry in the FAT structure on the disk

2

FAT File System Volume

- Four basic regions
  - 0 – Reserved Region
    - First disk sector
    - Boot sector, reserved sector, 0th sector
  - 1 – FAT Region
    - Always 2 FATs for redundancy
  - 2 – Root Directory Region (non FAT32)
    - Fixed size
  - 3 – File and Directory Data Region
    - Files are stored in clusters (cluster = N sectors)
    - Directories are specially formatted files

3

Boot Sector (FAT12,FAT16)

```
BYTE jumpBoot[3]        // jump instruction to boot code
BYTE SysName[8]         // name string (MSDOS5.0)
WORD BytesPerSector     // 512, 1024, 2048, 4096
BYTE SectorsPerCluster  // 1, 2, 4, 8, 16, 32, 64
WORD ReservedSectors    // 1 (for FAT12 and FAT16)
BYTE FATcount           // # of FAT data structures (always 2)
WORD MaxRootEntries     // # of 32-byte directory entries in root dir
WORD TotalSectors1      // Total sectors in the volume
BYTE MediaDescriptor    // Removable media is 0xF0
WORD SectorsPerFAT      // Sectors in ONE FAT
WORD SectorsPerTrack    //
WORD HeadCount
DWORD HiddenSectors
DWORD TotalSectors2
BYTE DriveNumber
BYTE Reserved1
BYTE ExtBootSignature
DWORD VolumeSerial      // Volume serial number
BYTE VolumeLabel[11]    // Volume label
BYTE Reserved2[8]
```

4

Disk Structure

Boot Sector	0
FAT 1	1 thru SectorsPerFAT
FAT 2	(SectorsPerFAT+1) thru (2*SectorsPerFAT)
Root Directory	(2*SectorsPerFAT +1) thru (2*SectorsPerFat+RootDirSectors)
File clusters	(2*SectorsPerFat+1+RootDirSectors) thru TotalSectors - 1

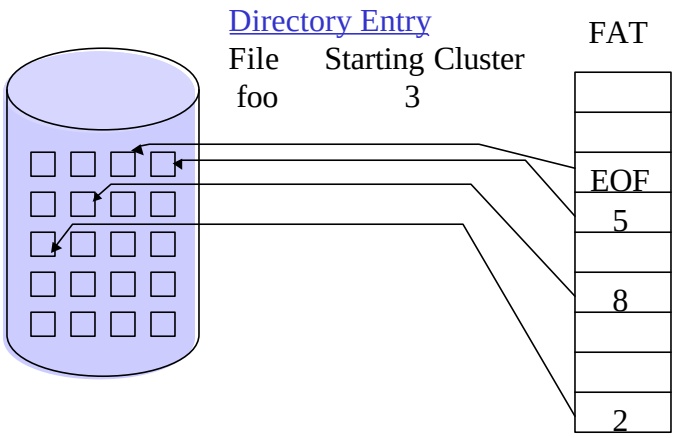
5

FAT Region

- There are `FATcount` (2) FATs in the volume
  - Keep both tables consistent
- Each FAT is `SectorsPerFat` sectors long
  - FAT12 lab – 9
- Each FAT is a singly linked list of the clusters of a file
- All linking is done using logical cluster numbers
- The first data cluster number is 2
- The first sector of cluster 2
  - RootDirSectors = ((MaxRootEntries*32)+(BytesPerSector-1))/BytesPerSector
  - FirstDataSector = ReservedSectors + (FATcount * SectorsPerFAT) + RootDirSectors
- The first sector of any cluster N
  - FirstSectorOfCluster = ((N-2) * SectorsPerCluster) + FirstDataSector

6

FAT File System



7

FAT12

- Each FAT12 entry is 12 bits
  - When designed, space was tight
  - Pack 2 entries into 3 bytes
  - 4096 possible clusters
  - If a sector is 512 bytes and cluster = 1 sector, can represent 2 Mb of data
- First 2 entries in the FAT are reserved
  - Don't use
  - Logical cluster numbers begin at 2
- FAT12 entry values
  - 0 Unused cluster
  - 0xFF0-0xFF6 Reserved cluster
  - 0xFF7 Bad cluster
  - 0xFF8-0xFFF End Of Clusterchain mark (EOC) <EOF>
  - Other Next cluster in file

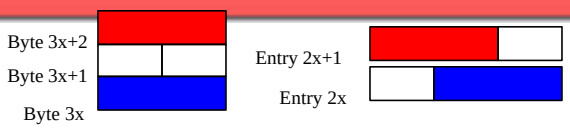
8

FAT12 entry packing

- Bytes 3x to 3x+2
  - contain FAT entries for logical clusters 2x and 2x+1
- Byte 3x
  - 8 least sig bits of entry 2x
- Byte 3x+1
  - 4 least sig bits are the 4 most sig bits of entry 2x
  - 4 most sig bits are the 4 least sig bits of entry 2x+1
- Byte 3x+2
  - 8 most sig bits of entry 2x+1

9

FAT12 entry packing



- Example FAT:
  - F0 FF FF 03 40 00 05 60 00 07 80 00 09 F0 FF
- First 3 bytes are first 2 entries → reserved
- Suppose first file cluster is 2 → bytes 3 and 4
- Converting entries 2 and 3
  - 03 40 00 → 003 and 004
- FF8 – FFF indicates EOF

10

Getting a FAT12 Entry

- Determine byte offset in the FAT
  - FatIndex = (LogicalCluster * 3)/2
  - FatIndex + 1
- Determine if an Even or Odd Entry
- Split LogicalCluster
  - Even → 4 most sig, 8 least sig
  - Odd → 8 most sig, 4 least sig
- Build next Logical Cluster number
  - Start with zero
  - Copy most sig bits and shift
  - Or in least sig bits

11

Get FAT12 entry (bytes)

```
int getCluster(int LogicalCluster)
{
    sector = 0
    // Convert LogicalCluster to byte location
    FatIndex = (LogicalCluster * 3)/2
    if (LogicalCluster is even)
       //split FatIndex + 1 for 4 most sig bits
       sector = FAT[FatIndex + 1] & 0x0f
       sector = sector << 8
       sector = sector | FAT[FatIndex]
    else
       //split FatIndex for 4 least sig bits
       sector = FAT[FatIndex+1]
       sector = sector << 4
       sector = sector | ((FAT[FatIndex] & 0xf0) >> 4)
}
```

12

Setting a FAT12 Entry

- Determine byte offsets in the FAT
  - FatIndex = (LogicalCluster * 3)/2
  - FatIndex + 1
- Determine if an Even or Odd entry
- Split LogicalCluster
  - Even → 4 most sig, 8 least sig
  - Odd → 8 most sig, 4 least sig
- Set FAT12 table entry

13

Set FAT12 entry (bytes)

```
void setFatEntry(int LogicalCluster, int value)
{
    // Convert LogicalCluster to byte location
    FatIndex = (LogicalCluster * 3)/2
    if (LogicalCluster is even)
       //split FatIndex + 1 for 4 most sig bits
       FAT[FatIndex+1] &= 0xf0
       FAT[FatIndex+1] |= (value & 0xf00) >> 8
       FAT[FatIndex] = value & 0x0ff
    else
       //split FatIndex for 4 least sig bits
       FAT[FatIndex] &= 0x0f
       FAT[FatIndex] |= (value & 0x00f) << 4
       FAT[FatIndex+1] = (value & 0xff0) >> 4
}
```

14

Byte Ordering

- Intel and FAT byte ordering is Little Endian.
- FAT12 table in Hex
  - each byte given in Little Endian order
    - F0 FF FF 03 40 00 05 60 00 07 80 00 09 F0 FF
  - Ordering nibbles (half-byte) from least to most significant
    - 0F FF FF 30 04 00 50 06 00 70 08 00 90 0F FF
- The Little Endian arrangement of bytes makes it convenient to extract and set the entries using a byte offset to the WORD containing the entry in the table
  - An entry is the least significant (Even) or most significant (Odd) 12 bits of the WORD

15

Get FAT12 Entry (WORDs)

```
FATEntOffset = (ClusterNum * 3)/2
Fat12ClusterEntry = *((WORD*) &FATbyte[FATEntOffset]);
If(ClusterNum & 0x0001)
        // entry is the high order 3 bytes of the WORD
        Fat12ClusterEntry = Fat12ClusterEntry >> 4;
Else
        // entry is the low order 3 bytes of the WORD
        Fat12ClusterEntry = Fat12ClusterEntry & 0x0FFF;
```

16

Set FAT12 Entry (WORDs)

```
FatEntOffset = (ClusterNum * 3)/2
if (ClusterNum & 0x0001)
        // entry is the high order 3 bytes of the WORD
        Fat12ClusterEntry = Fat12ClusterEnty << 4;
        *((WORD*) &FAT[FatEntOffset] =
                *((WORD*)&FAT[FatEntOffset] & 0x000F;
Else
        // entry is the low order 3 bytes of the WORD
        Fat12ClusterEntry = Fat12ClusterEntry & 0x0FFF;
        *((WORD*)&FAT[FatEntOffset] =
                *((WORD*)&FAT[FatEntOffset] & 0xF000;

*((WORD*) &FAT[FatEntOffset] =
        (*((WORD*) &FAT[FatEntOffset])) | Fat12ClusterEntry;
```

17

Typical FAT12 Floppy Disk

- 1 sector per cluster
- 1 sector reserved (Boot sector)
- 18 sectors for 2 FAT tables
- 14 sectors for root directory
- Convert logical cluster number to sector number
  - Subtract 2 from logical cluster number
  - Multiply by number of sectors per cluster (1)
  - Add result to first data area sector number (33)

18

Directories

- Array of directory entries
- Root directory has a fixed number of entries
  - 14 sectors reserved → How many entries?
  - Contiguous sectors
- Subdirectories are simply files
  - Same array of directory entries structure within each cluster (no size restriction)
- Directory entry
  - 32 bytes
  - Filename's first character is usage indicator
    - 0x00 Never been used
    - 0xe5 Used before but entry has been released

19

Directory Entry

```
BYTE Filename[8]     // Pad with spaces
BYTE Extension[3]    //
BYTE Attributes
BYTE Reserved[10]
WORD Time            // Hour*2048+Min*32+sec/2
WORD Date            // (Yr-1980)*512 + Mon*32 + day
WORD StartCluster
DWORD FileSize
```

20

File Attributes

Bit	Mask	Attribute
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Unused
7	0x80	Unused

21

Suggested Lab Progression

- Read boot sector and print out
- Read root directory and print out
- Read FAT and interpret for files in root dir
  - Print out cluster list for each file
- Read in and print out a file
- Read in and print out a subdirectory
- Finish up
- Make DLL

22