Data Recovery on The Hard Disk in FAT32 Format

Nguyen Tien Duy
Thai Nguyen University of Technology
Thai Nguyen University
Thai Nguyen, Vietnam
duynt@tnut.edu.vn

***Abstract*—Data is a very important component of computing systems. Data storage usually files in many formats. During the process of using, storing, copying and transporting data files, it is inevitable for them to accidentally delete data. In this situation, being able to "retrieve" those data files is essential. Originating from that essential need, this paper has studied the logical structure of hard disk in FAT32 format, mechanism of storing and managing data on disk, and how to delete a data file on disk. Since then proposed algorithms for recovering deleted data. Utility software according to the proposed algorithm has also been installed. The test results show that the ability to recover deleted files is very good.**

Keywords—FAT32, Lost and Found, Data Recovery, Get Data Back

I. INTRODUCTION

Data is a representation of the information, in fact, the information exists very diverse so the data is also very diverse. In the strong development of automatic information processing on computers, data needs to be shared for many users, multiple uses. That means data needs to be communicated, stored and processed. In most systems today, the value of the data is very large, many times greater than the cost of the hardware system. For example, data stored in sectors such as military, banking, meteorology, etc. or for every business or individual, the data is of great value and importance. With the importance of data, depending on the different systems, one needs to back up the data at different times so that in case of loss of data in the process of working, there is a backup copy [1], [2].

When working with data, a common problem for computer users is losing data (for some reason, often erasing) on storage devices, especially losing data on the hard disk. When users want to find their important data in many ways, as quickly as possible. It's not that backing up data is the perfect solution, in many situations where there is no backup, the problem becomes serious. An effective measure, in this case, requires a utility tool that allows the above mentioned problem to be solved [5].

In this study, we study the physical structure, logical structure of hard magnetic disk in FAT32 format, storage mechanism and data management on disk. Since then the proposed data recovery algorithm has been deleted. Algorithm installation and software testing were also carried out. Test results show that the ability to recover data is quite good in case the hard drive is not fragmented. When there is fragmentation, resilience is not completely.

II. MANAGE FILES ON THE HARD DISK IN FAT32 FORMAT

A. *Boot Sector (FAT Partition Boot Sector)*

The Boot Sector contains information that the file system uses to access the volume. On x86 computers, the MBR uses the Boot Sector on the system partition to load the operating system kernel files. The Table 1 describes the information fields in the Boot Sector with a FAT file system formatted Volume.

TABLE I. INFORMATION FIELDS IN BOOT SECTOR

Byte Offset (in hex)	***Field Length***	***Sample Value***	***Meaning***
00	3 bytes	EB 3C 90	Jump instruction
03	8 bytes	MSDOS5.0	OEM Name in text
0B	25 bytes		BIOS Parameter Block
24	26 bytes		Extended BIOS Parameter Block
3E	448 bytes		Bootstrap code
1FE	2 bytes	0x55AA	End of sector marker

The Table 2 describes the fields in the BIOS parameter block and the expanded BIOS parameter block.

TABLE II. BIOS PARAMETER BLOCK AND EXTENDED BIOS PARAMETER BLOCK FIELDS

Byte Offset	Field Length	Sample Value	Meaning
0x0B	WORD	0x0002	Bytes per Sector. The size of a hardware sector. For most disks in use in the United States, the value of this field is 512.
0x0D	BYTE	0x08	Sectors Per Cluster. The number of sectors in a cluster. The default cluster size for a volume depends on the volume size and the file system.
0x0E	WORD	0x0100	Reserved Sectors. The number of sectors from the Partition Boot Sector to the start of the first file allocation table, including the Partition Boot Sector. The minimum value is 1. If the value is greater than 1, it means that the bootstrap code is too long to fit completely in the Partition Boot Sector.
0x10	BYTE	0x02	Number of file allocation tables (FATs). The number of copies of the file allocation table on the volume. Typically, the value of this field is 2.
0x11	WORD	0x0002	Root Entries. The total number of file name entries that can be stored in the root folder of the volume. One entry is always used as a Volume Label. Files with long filenames use up multiple entries per file. Therefore, the largest number of files in the root folder is typically 511, but you will run out of entries sooner if you use long filenames.
0x13	WORD	0x0000	Small Sectors. The number of sectors on the volume if the number fits in 16 bits (65535). For volumes larger than 65536 sectors, this field has a value of 0 and the Large Sectors field is used instead.
0x15	BYTE	0xF8	Media Type. Provides information about the media being used. A value of 0xF8 indicates a hard disk.
0x16	WORD	0xC900	Sectors per file allocation table (FAT). Number of sectors occupied by each of the file allocation tables on the volume. By using this information, together with the Number of FATs and Reserved Sectors, you can compute where the root folder begins. By using the number of entries in the root folder, you can also compute where the user data area of the volume begins.
0x18	WORD	0x3F00	Sectors per Track. The apparent disk geometry in use when the disk was low-level formatted.
0x1A	WORD	0x1000	Number of Heads. The apparent disk geometry in use when the disk was low-level formatted.
0x1C	DWORD	3F 00 00 00	Hidden Sectors. Same as the Relative Sector field in the Partition Table.
0x20	DWORD	51 42 06 00	Large Sectors. If the Small Sectors field is zero, this field

Byte Offset	Field Length	Sample Value	Meaning
			contains the total number of sectors in the volume. If Small Sectors is nonzero, this field contains zero..
0x24	BYTE	0x80	Physical Disk Number. This is related to the BIOS physical disk number. Floppy drives are numbered starting with 0x00 for the A disk. Physical hard disks are numbered starting with 0x80. The value is typically 0x80 for hard disks, regardless of how many physical disk drives exist, because the value is only relevant if the device is the startup disk.
0x25	BYTE	0x00	Current Head. Not used by the FAT file system.
0x26	BYTE	0x29	Signature. Must be either 0x28 or 0x29 in order to be recognized by Windows NT.
0x27	4 bytes	CE 13 46 30	Volume Serial Number. A unique number that is created when you format the volume.
0x2B	11 bytes	NO NAME	Volume Label. This field was used to store the volume label, but the volume label is now stored as special file in the root directory.
0x36	8 bytes	FAT16	System ID. Either FAT12 or FAT16, depending on the format of the disk.

B. Files and Folders were deleted

When we delete a file, the operating system does not actually delete its data, it does the following two things:

- Put the first byte in the corresponding entry with 0xE5 (229).
- Delete 0x00000000 values of all FAT entries corresponding to the file's supply chain (32 bits value).

In case the object to be deleted is a folder, DOS deletes the objects in it first and then deletes the folder.

Thus, in essence, the information content of a file (or directory) remains intact on its clusters. The operating system considers these clusters to be free because the entries corresponding to them in FAT are 0. It can be re-allocated to other files when creating and writing files to disk, new information will be overwritten onto the old information.

If the supply chain of a deleted file has a serial number, based on the cluster number starting in the directory entry and the size of the file, it is possible to recover this deleted file. By modifying the first byte of the directory entry to a value other than 0x2E and recording the FAT entry sequence starting from the entry with the serial number in the directory entry with the successive value, the last FAT entry is recording the value indicating the end (the number of clusters of files can be calculated based on the size of the file).

III. RECOVER THE FILES AND FOLDERS WERE DELETED

A. Algorithm

Based on the structure and storage organization of the hard disk formatted in the FAT file system, we can build an algorithm to search for deleted files and perform read data of a file to record it as a new file, here we just copy the data from the deleted file into a new file "image" with its format exactly in a binary format without knowing the source that created it. In principle, it is possible to recover files by modifying Root and FAT, when the file is restored to its original location when not deleted, but writing the information to the system area of the drive Root and FAT is not recommended. If the amendment is incorrect, the consequence is that it will not only recover data but also affect other files (folders). The solution to copy the data is the securest and chosen in this case. Specifically, simplify the implementation steps of the algorithm as follows:

Read_DBR() // Determine the parameters of the drive
Output:

- bytes/sector
- sectors/cluster
- number of FAT
- number of sectors/root
- number of sectors/FAT
- First cluster of Root

- First cluster of Data

End of Read_DBR
Read_Cluster_Root()
Output:

- List of deleted files.
- Allow users to select files to be recovered.
- Open a new file
- Get the size file, determine the number of clusters (k).
- Get the starting cluster number of the file.

Loop:

- Read data
- Write to the file
  Read FAT, determine the next cluster number.
  Until the value entry_FAT<>0 // Fragments
- When entry_FAT<>0, next //Jump over the fragmented area
- Goto Read FAT

End_Loop

- Close file

End of Reeead_Cluster_Root

B. Fragmentation and impact on data recovery

In fact, the array clusters supply for a file is often discontinuous, especially when the file is big. This phenomenon is called a fragment of the file. Fragmentation creates during successive creation and deletion of files (folders) while working. In addition, for multitasking operating systems, it uses the free part of the hard disk during an operation to implement the virtual memory mechanism cause to fragmentation (we do not go into this issue).

When formatting, copying data to the drive and installing the entire system, there was no fragmentation. Increasing fragmentation as the work process. Even then, there may be files that are stored in different sectors on the disk. Access to the information of the fragmented files will take longer than the non-fragmented files (because it must control the read/write head to the disk areas with very far coordinates), so after each working time of the drive hard drive, we should “defragment” with the built-in utility of Windows - Defragment utility.

When fragmentation, the recovery of deleted files has many difficulties, making the probability of recovery is not 100%, even many cases can not be restored.

Assuming a file is written to disk, the operating system will provide free space for this file. However, these empty areas are much smaller than the file size. Therefore, this file is fragmented into several segments, interwoven with it as other files. When this file is deleted, if the segments interwoven with it are being used (or bad), not in a free state, then the chances of re-reading information from the sections of the deleted file are large. On the contrary, the interleaved sections with the file deleted in this free state (the file interwoven with it have also been deleted), it is very difficult to determine which of these free join sectors belong to which file. We are in need of restoration because the information about these complexes is zero in the FAT table.

However, in many situations, it is valuable to save some deleted data.

C. Installation and testing

To coding the above algorithm, we can use certain programming languages among many possible languages.

- Assembly is a low-level language, showing the advantage of being a compact target code, the program runs fast. In the coding process of the algorithm, calling DOS and BIOS services is very intuitive. However, when solving a problem of complexity and relative magnitude, this language reveals many disadvantages: creating an elaborate interface, difficult to control algorithms and generally takes a lot of effort of the programmer.

- Languages like Pascal and C are high-level languages that allow for deep intervention into the system, making interrupt calls (DOS and BIOS services) easy. These languages are highly scholarly, close and simple with an algorithm coding. Pascal is tightly structured, along with C, these are familiar languages for students and most people who know how to program. One drawback of these languages is the elaborate interface creation, the program may not be compatible in some Windows environments.

In this study, I chose Pascal as a programming language to encode algorithms to suit the majority of students who want to learn about this problem. Some data structures and procedures are presented in the Appendix.

Testing the program has been done many times with different sized files. Most recently deleted files can be recovered almost completely. In some situations of creating and deleting files after deleting files to recover, the ability to recover is low. Especially when the hard drive is in a fragmented state, the recovery ability is not high.

IV. CONCLUSION

Based on a study of the structure and organization of data storage from a hard disk formatted according to the FAT file system and the meaning of data recovery. This study has achieved certain results. In many experiments with data files of sizes, fragmentation, the program almost recovered data.

This study was only installed on hard drives with the FAT file system. Next, we will develop this program to a more complete level, expand the NTFS file system format, thoroughly overcome the existing problems, allowing data recovery in many logical drives on one

machine single or run on the network to recover data from another device.

ACKNOWLEDGMENT

This research was funded by the science and technology fund of the Thai Nguyen University of Technology (TNUT).

REFERENCES

[1] William Stallings; Computer Organization and Architecture Designing for Performation; Prentice Hall.

[2] Shri Vishnu Engineering College for women:: Bhimavaram Department of Information Technology, "Computer Organization and Architecture lecture notes".

[3] "Design of the FAT file system", https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

[4] Michael Hordeski, Personal Computer Interfaces, Mc. Graw Hill, 1995.

[5] Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall, 1996.

[6] S.I.Ahson, "Microprocessor with application in Process Control", Tata Mc.Graw Hill, 1984.

[7] Gustaf Olsson, Gianguido Piani, "Computer Systems for Automation and Control", Prentice Hall, 1990.

[8] Mikell P.Groover, Automation, Production System and Computer Integrated Manufacturing, Prentice Hall.

[9] "Complete info source: NTFS & FAT file systems and data recovery", https://www.ntfs.com/index.html

APPENDIX: SOURCE PROGRAM

When implementing the algorithm, several main data structures and procedures are constructed as follows:

Data structure:

```
Type
    Package = record
              size: byte;
              reserved1: byte;
              NumSec: byte;
              reserved2: byte;
              offset: word;
              segment: word;
              LBA: array[0..1] of longint
    end;
    Buffer = record
           BS_jmpBoot: array[0..2] of byte;
           BS_OEMName: array[0..7] of char;
           BPB_BytsPerSec: word;
           BPB_SecPerClus: byte;
           BPB_RsvSecCnt: word;
           BPB_NumFats: byte;
           BPB_RootEntCnt: word;
           BPB_TotSec16: word;
           BPB_Media: byte;
           BPB_FATSz16: word;
           BPB_SecPerTrk: word;
           BPB_NumHeads: word;
           BPB_HiddSec: longint;
           BPB_TotSec32: longint;
           BPB_FATSz32: longint;
           BPB_ExtFlags: word;
           BPB_FSVer: word;
           BPB_RootClus: longint;
           BPB_FSInfo: word;
           BPB_BkBootSec: word;
           BPB_Reserved: array[0..11] of byte;
           BS_DrvNum: byte;
           BS_Reserved1: byte;
           BS_BootSig: byte;
```

```
            BS_VolID: longint;
            BS_VolLab: array[0..10] of char;
            BS_FilSysType: array[0..7] of char;
            BS_R: array[0..421] of byte
    end;
    Sector = array[0..511] of byte;
    Cluster = array[0..7] of Sector;
```

Procedures:

```
procedure read_dbr;
Begin
     with DAP do
     begin
          size:=$10;
          reserved1:=0;
          NumSec:=1;
          reserved2:=0;
          offset:= ofs(dbr);
          segment:=seg(dbr);
          LBA[0]:=63;
          LBA[1]:=0
     end;
     with r do
     begin
          ax:=$4200;
          dl:=$80;
          ds:=seg(DAP);
          si:=ofs(DAP)
     end;
     intr($13, r);
End;

procedure read_sector(num_sec: longint; var buffer: Sector);
Begin
     with DAP do
     begin
          size:=$10;
          reserved1:=0;
          NumSec:=1;
          reserved2:=0;
          offset:= ofs(buffer);
          segment:=seg(buffer);
          LBA[0]:=num_sec+63;
          LBA[1]:=0
     end;
     with r do
     begin
          ax:=$4200;
          dl:=$80;
          ds:=seg(DAP);
          si:=ofs(DAP)
     end;
     intr($13, r);
End;

procedure read_cluster(num_clus: longint; var buffer: Cluster);
var
   i: byte;
   num_sec: longint;
Begin
     num_sec:=((num_clus-2)*dbr.BPB_SecPerClus+FirstDataSector;
     for i:=0 to 7 do
         read_sector(num_sec+i, buffer[i]);
End;
```

```
file_size:=meml[seg(data_clus[k,i*32+28]):ofs(data_clus[k,i*32+28])];
Entry_Num:=round(file_size/(dbr.BPB_BytsPerSec*dbr.BPB_SecPerClus))+1;
Clus_NumH:=memw[seg(data_clus[k,i*32+20]):ofs(data_clus[k,i*32+20])];
Clus_NumH:= (Clus_NumH shl 16);
Clus_NumL:= memw[seg(data_clus[k,i*32+26]):ofs(data_clus[k,i*32+26])];
First_Clus_Num:=Clus_NumH+Clus_NumL;
writeln('Fisrt Cluster: ', First_Clus_Num);
next_clus:=First_Clus_Num+1;
entry_fat:=0;
assign(f, 'C:\recover.bak');
rewrite(f);
k:=0;
while (k<Entry_Num) do
begin
      repeat
         read_cluster(next_clus, data);
         for i:=0 to 7 do
         for j:=0 to 511 do
         write(f, data[i, j]);
         FATOffset:=next_clus*4;
         ThisFATSecNum:=dbr.BPB_RsvSecCnt+
                          (FATOffset div dbr.BPB_BytsPerSec);
         ThisFATEntOffset:=FATOffset mod dbr.BPB_BytsPerSec;
         read_sector(ThisFATSecNum, fat);
         entry_fat:=meml[seg(fat[ThisFATEntOffset]):
                                              ofs(fat[ThisFATEntOffset])];
         k:=k+1; next_clus:=next_clus+1;
      until (entry_fat<>0);

      while (entry_fat<>0) do
      begin
         FATOffset:=(next_clus)*4;
         ThisFATSecNum:=dbr.BPB_RsvSecCnt+
                               (FATOffset div dbr.BPB_BytsPerSec);
         ThisFATEntOffset:=FATOffset mod dbr.BPB_BytsPerSec;
         read_sector(ThisFATSecNum, fat);
         entry_fat:=meml[seg(fat[ThisFATEntOffset]):
                                              ofs(fat[ThisFATEntOffset])];
         inc(next_clus)
      end
end;
close(f);
```