Full Name:

- Nguyễn Công Anh Khoa - 18127261

- Vũ Công Minh - 18127155

- Lê Minh Thanh Tú – 18127271

# Report File System

## I. Structure of Volume:

Volume is separate into cluster. Each cluster separates into sector. Cluster is counted form 0.

Vol has two area, system area and data area.

System area contain information to manage volume. Data area contain information of file.

System area has 3 part. Firstly, Boot sector contain information about disk (Bytes per Sector, Sector per Cluster, Total Sector, …). Secondly, Cluster manager contain information to manage cluster (free cluster or is using cluster). Finally, Record manager contain necessary information to access file or folder. Space of system area is about 13% of volume

|  |  |  |  |
| --- | --- | --- | --- |
| Boot Sector | Cluster manager | Record manager | Data |

### Boot Sector

Always starts at the head of volume. Contain information about Volume

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 8 bytes | String “ AKCMMT “ |
| 08h | 4 bytes | Bytes per Sector |
| 0Ch | 4 bytes | Sector per Cluster |
| 10h | 8 bytes | Total Sector |
| 18h | 4 bytes | Sector begin of Cluster manager |
| 1Ch | 4 bytes | Sector begin of Record manager |
| 20h | 48 bytes | String “Copyright belong to Anh Khoa vs Cong Minh vs Minh Tu” |
| 54h | 2 bytes | End marker AA 55 |

*Layout of Boot sector*

### 2. Cluster manger

Cluster manager begin at sector 2. Size of Cluster depend on how many clusters in volume. Total cluster in volume is calculated by get total sector / sector per cluster.

Cluster manager contain sequence of bit, each bit represent cluster. 0 is free cluster, 1 is cluster is using. We group each 8 adjacent cluster, so each byte represent state of 8 adjacent cluster. We access state of S’th cluster in cluster manager by this way:

Step 1: Go to offset S / 8 of cluster manager.

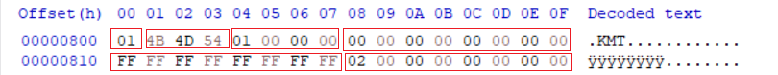
Step 2: Go to S % 8 bit of byte in this offset, it is bit represent of S’th cluster.

### 3. Record manager

Each file or folder holds one or some record. Each record hold entire 1 sector, we enumerate record follow sector contain it. If file holds some record, we use structure link list to manage record. We also use structure tree to manage folder and file, each folder and file has only one parent folder. First record of file is called file’s base record. Each record begins with record header, then attributes of file.

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 1 byte | Flag: 1 means file is using, 2 means folder is using, 0 means empty record |
| 01h | 3 bytes | String “KMT” |
| 04h | 4 bytes | Number of records |
| 08h | 8 bytes | Sector of record’s parent (00 00 00 00 00 00 00 00 when it is the first record) |
| 10h | 8 bytes | Sector of next record (FF FF FF FF FF FF FF FF when it is the last record) |
| 18h | 8 bytes | Sector of parent folder (00 00 00 00 00 00 00 00 when it doesn’t belong to any folder) |

*Layout of Record header*

**

*Example of Record Header*

Attribute of file comprise File Info (Information about file), Security (password of file), Data (cluster that file hold, just for file), and Index (a list of file attribute of file or subfolder within folder).

Attribute is represented by a number

|  |  |
| --- | --- |
| Attribute type | Description |
| 10h | File Info |
| 20h | Security |
| 30h | Data |
| 40h | Index |
| FF FF FF FFh | End marker |

Each attribute begins with attribute header.

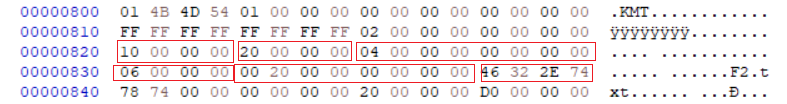
|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 4 bytes | Attribute type |
| 04h | 4 bytes | Attribute length (if we seek Attribute length bytes, we will begin of next attribute) |

*Layout of attribute header*

1. File Info
2. Keep information about file name, sector of base record of file. File Info attribute always is held entirely in base record. We just support name of file in utf8

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 8 bytes | Sector of base record |
| 08h | 4 bytes | Length of file name |
| 0Ch | 8 bytes | Size of file in bytes (0 with folder) |
| 14h | Length of file name | Name of file |

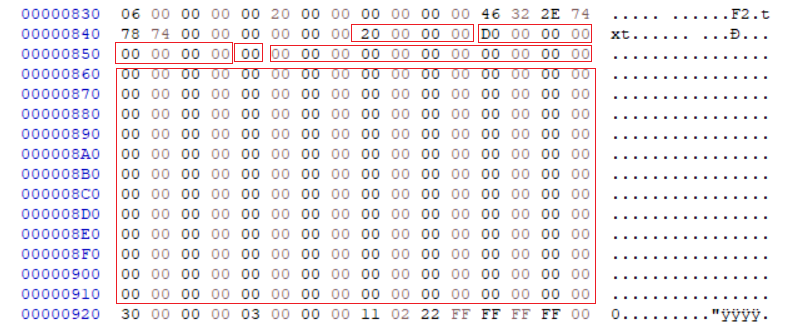
*Layout of File Info*

**

*Example of File Info Attribute*

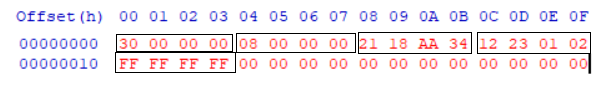
1. Security
2. Contain information about password of file. Password is a number with at most 8 digits. Just likes File Info, Security is held entirely in base record

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 4 bytes | Length of password has been hashed (0 if doesn’t have password) |
| 04h | 1 byte | Flag: 0 means doesn’t have password, 1 means other wise |
| 05h | 200 bytes | Buffer of password |

*Layout of Security*

*Example Of Security Attribute*

1. Data
2. Keep information about clusters that contain data of file. It can’t be held enough in one record, it will be stored in some sub record if too much cluster. For example, if data store in cluster 1, 2, 3, 6, 7, 8. Cluster 1, 2, 3, have been stored in data attribute if base record and didn’t have any space for storing cluster 6, 7, 8, we will use another record (is called sub record) and this record has structure similar base record but doesn’t have File Info, Security attribute, just have Data attribute and store cluster 6, 7, 8 in this attribute. We also compress adjacent clusters by using structure data run. Data run begin with a byte. The higher 4 bits is S, the lower 4 bits is L, the next L bytes represent length of adjacent clusters, the next S bytes represent cluster begin of this sequence clusters. For instance, we have a data run 21 18 AA 34, it represents sequence adjacent clusters begin in 34 AAh’th cluster and last 18h. We continue reading data run until end of data attribute (means the length of data attribute fits with length of attribute in attribute header).
3. Another example about data attribute, look at the follow picture



First 8 bytes is data attribute header, it tells we know this is data attribute (30 00 00 00h) and it lasts 8 byte (08 00 00 00h), the next 4 bytes is a data run 21 18 AA 34 which lasts 4 bytes, the next 4 bytes is the other data run 12 23 01 02, so we have read 8 bytes of data run, equal length of length of attribute in attribute header so we have read all data run in this record (the evident is end marker of attribute FF FF FF FFh)

If file has many clusters so we use sub record to store these clusters, we know the position of the next record by looking at sector of next record in record header.



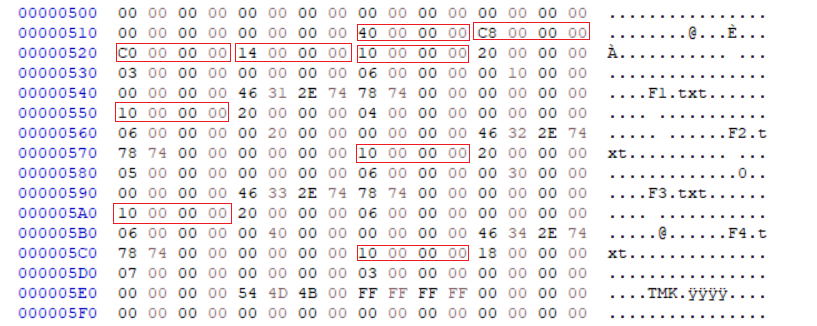
*Example of Data Attribute*

1. Index
2. Keep File Info of file or sub folder in folder. Index attribute like Data attribute, can be held in some record. After Index attribute header, it is Index header.

|  |  |  |
| --- | --- | --- |
| Offset | Size | Description |
| 00h | 4 bytes | Length of all File Info of file or subfolder in record |
| 04h | 4 bytes | Empty bytes remain in record |

*Layout Index header*

After Index header is File Info of file or subfolder of folder. If too much file or folder in folder, so we have to use sub record to store. The way we read the other Index attribute of folder is similar to the way we read data run of file.



*Example of Index Attribute*

Comment:

* Advantage:
* Manage Cluster more efficient than FAT, just cost 1 bit instead 1, 2, 4 bytes in FAT 8, 16, 32 to know state of cluster, and use structure data run to represent cluster is using (means we ignore cluster is free) and reduce space of adjacent cluster
* Don’t cost at least a cluster for folder
* Not a complex system, easy to implement
* Disadvantage:
* File Info isn’t complete, doesn’t contain attribute of file or folder like time create, time update, read only, hide, … (we have thought about it but don’t enough time to build)
* If we save many small files, it can cause run out of record, because all record of file or folder store in Record Manager which has fit size. We can improve it by allowing user to change system area of they volume (We can do this but don’t have enough time)
* We just support utf\_8 character, because if we support utf\_16 character, File Info and Security attribute can’t be stored in one record, it make so hard for us to implement (we still can do it if we have enough time)

## II. Operator of Volume

1. Create volume

Step 1: Open Volume

Step 2: Create Boot Sector from input Volume size

Step 3: Create an empty cluster with 512 bytes ‘00’

Step 4: Calculate the number of Cluster and write to file n (number of cluster) empty clusters

Step 5: Seek to the begin of file and write Boot Sector

Step 6: Create Cluster Manager and write it

Step 7: Close Volume

1. Import file
2. Open file needs to import
3. Location folder where file is imported
4. Read all file
5. Get the largest free sequence free clusters, if run out of cluster, return announce can’t add more file
6. Write (remain) data to these clusters
7. Maintain clusters use to store data
8. If don’t reach the end of data, return step 3
9. Get an empty record, if run out of record, return announce can’t add more file
10. Maintain record is use for this file
11. Write attribute File Info, Security, Data (for file), end marker to record in order, if reach end of record, return step 5 and continue write the content of attribute is writing
12. Link record headers in order (sector of record’s parent, sector of next record, number of record)
13. Write record headers
14. Update clusters has used to store file form free to is using
15. return announce successfully import file

Comment:

* Advantage:
* The way we use cluster, for us, we think it work efficiently, it using approach greedy to avoid fragment as much as possible, may be it doesn’t work efficiently in some case, but we think it work well in most case
* Disadvantage:
* We have to cost memory and time as large as volume size to find the largest adjacent clusters and, we improve performance by using priority queue, it work well when we frequently import file, but it work worse when we frequently delete file because we have to rebuild this queue after delete file. But we satisfy with it because it helps to reduce fragment in volume, which works so slowly and we maintain it in RAM, which works faster than disk. If we have more time, we will think about how to reduce memory and time to find the largest adjacent clusters
* Records of file may not be adjacent, so we have to seek to each record to get entire file information
* Some record still have many free bytes, some record doesn’t have enough bytes and have to need some sub record, but we satisfy with it because it easy to manager and we have used data run to compress clusters
* Don’t check if there is a file has same name

1. Import folder
2. Open folder needs to import
3. Get an empty record, if run out of record, return announce can’t add more folder
4. Write header attribute File Info, Security, Index attribute to record
5. Get path of file or subfolder of folder, if has import all file or subfolder, return announce import folder successfully
6. Define if this path is a path of folder or not
7. If this path is path of folder, call function import folder
8. If this path is path of file, call functions import file
9. Get File Info attribute of file or folder has just imported
10. Travel from all record of folder
11. If have one of record in these records has enough empty bytes to store this File Info attribute, store this File Info attribute to this record, return step 4
12. Otherwise, get another empty record, if run of record return announce can’t add more folder
13. Write record header, Index attribute and add File Info attribute into this Index attribute
14. Update all record header of folder, return step 4

* Advantage:
* Don’t cost at least one cluster to maintain file and subfolder
* Use tree structure so it easy to find file and folder in volume
* Disadvantage
* Just like file, record of folder may not be adjacent and some record has many empty bytes, some other not
* Don’t check if there is a folder has same name

1. Delete file
2. Step1: Create a list of entries of the file that you want to delete, this list will store entries to change it ( change to 0) later.
3. Use function SeekToSector to move the pointer to the file entry that you want to delete.
4. Use function read of RecordHeader class to skip through the file’s header.
5. Then we want to skip the File Name Attribute and Security Attribute by using function read of AttributeHeader (to read and move the pointer to the end of the Attribute Header), and seekg of Vol (to skip through the information of these Attributes).
6. We skip through the Data Attribute as well, by using the same way in the previous step.
7. We need to release Datarun in Cluster Manager. We use 1 while loop, which the condition is AlreadyRead < Size (Size = Data length). First, we read the data by using the function read. Then increase the AlreadyRead equal to the datarun size. Next, we use update function to update the cluster of the data in the Cluster Manager.
8. Then we check if this file has more than 1 entry, if it does, we do the same as the 1st entry but this child entry doesn’t have the File Name and Security Attribute. And we also store this child entry into the list in step 1.
9. Then we update the entry we had stored in the list of entries in step 1, we seek to it then change it value to 0. We also mark this entry is free by push it into FreeEntry list.
10. Finally, we check if this file is in a folder or not, if yes, we call function DeleteFileInfoInFolder, this function will delete file’s entry in it’s parent folder’s index list.
11. Delete folder
12. Create a list of entries of the folder that you want to delete, this list will store entries to change it (change to 0) later. And a list to store entries of un-deleted file or folder.
13. Use function SeekToSector to move the pointer to the folder entry that you want to delete.
14. Use function read of RecordHeader class to skip through the folder’s header.
15. Then we want to skip the File Name Attribute and Security Attribute by using function read of AttributeHeader (to read and move the pointer to the end of the Attribute Header), and seekg of Vol (to skip through the information of these Attributes).
16. We skip through the Index Attribute as well, by using the same way in the previous step.
17. We need to read the Index data to store every element of the Index list in the FileOrFolder list. By read check every type of the element (if it’s different than 229, it means that element hadn’t been deleted yet, then we push it into FileOrFolder List).
18. Then we check if this Index List has more than 1 entry, if it does, we do the same as the 1st entry but this child entry doesn’t have the File Name and Security Attribute.
19. We update the entry we had stored in the list of entries in step 1, we seek to it then change it value to 0, it mean it had been deleted.
20. We check every element in the FileOrFolder list, depend on the element’s type, we call function DeleteFile or DeleteFolder.
21. Finally, we check if this folder is in a folder or not, if yes, we call function DeleteFileInfoInFolder, this function will delete folder’s entry in it’s parent folder’s index list.
22. Move file

Step1: Use entry file to seek to its header

Step2: Read and move to the end of RecordHeader

Step3: Read and move to the end of filename attribute (first attribute)

Step4: Now we are pointing to FIleInfo attribute

Step5: Get FileSize and FileName, then seek to the end of FIleInfo attribute

Step6: Read and move to the end of Security attribute and datarun attribute

Step7: Now, read all dataruns to a list

Step8: Use each dataruns to read length and offset of data, seek to its offset and write data to a new file

Step9: Delete our file

1. Move folder

Similar to MoveFile function, but you have to call recursively its subfolders and subfiles

Step1: Read and pass attributes, just save folder size and folder name.

Step2: Create list of datarun and use it to reach current folder’s data

Step3: Call MoveFolder when our pointer point to subfolder, call moveFile when it point to subfile. When you call movefolder for subfolder and subfile, remember to add “\\” to its path before.

Step4: Delete our folder

1. Create pass

Step1: Use entry file to seek to its header

Step2: Read RecordHeader and move to the end of file’s header

Step3: Read AttributeHeader and move to the end of FileInfo Attribute Header

Step4: Seek to Password part and encrypt from a string input

Step5: Read AttributeHeader and move to the end of Security Attribute Header

Step4: Write info to Password part.

\*\* We have function HasPass to check if our file has a password, it will return true and the password after decrypting, else return false and empty password.

## III. Algorithms

1. **Encrypt password**

Firstly, your password contains 8 numbers (0-9). For example: (PassArray)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 8 | 2 | 4 | 0 | 0 | 7 | 5 |

Secondly, random choose a permutation of 10: For example: (PerArray)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 4 | 3 | 5 | 2 | 8 | 1 | 6 | 7 | 0 |

Finally, we have a special array which will be stored in our data: (SpecialArray[i] = PerArray[PassArray[i]])

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 | 3 | 2 | 9 | 9 | 6 | 8 |

We write on our data perArray and SpecialArray (10 + 8 = 18 numbers)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 4 | 3 | 5 | 2 | 8 | 1 | 6 | 7 | 0 | 3 | 7 | 3 | 2 | 9 | 9 | 6 | 8 |

Now, we have 18 blocks. Each block has 3 parts: data, next index, trash.

For example: red is data, blue is address, black is trash

Block 1: random the block length = 5 => next index of block 2 is 8

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 09 | 05 | 06 | 03 | 08 |

0 1 2 3 4

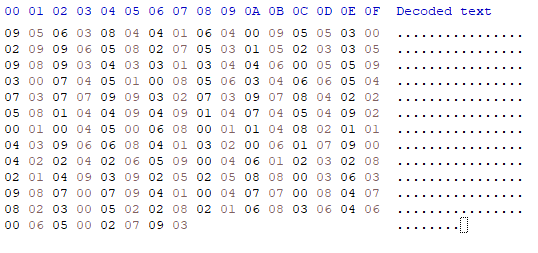
Block 2: random trash length = 3 => next index of block is 14

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 09 | 05 | 06 | 03 | 08 | 04 | 04 | 01 | 06 | 04 | 00 | 09 | 05 | 05 |

0 1 2 3 4 5 6 7 8 9 10 11 12 13

\*If next address is an integer contains 2 characters, it will be stored in 2 offset (14 is stored in offset 6 and 7)

… After all, we have all 18 unfixed size blocks. Then we add some trash to fill 200 characters at the end



- Advantages:

+ With the same input, we have many different ways to create perArray, to create blocks, … => Length of real data blocks are unfixed

+ Trash byte is modified as the same with data and index => obscuration

- Disadvantages:

+ Input contains 8 basic characters (0-9) => Limit on test case

+ Input password still fixed (8 integers)

1. **Decrypt password**

* Firstly, read 200 characters from our data memory.
* To decrypt our password, we have to read two arrays: perArray and specialArray
* We have: perArray has 10 blocks, specialArray has 8 blocks => Read data of 18 blocks => use a tempArray[18]
* For example, we can see first 16 characters:



* Each block has data at the first => Read first character into tempArray[0] => tempArray[0] = 3
* Next characters are next address of tempArray[1] => tempArray[1] is stored at 8
* Then, skip trash characters of block 1, focus on block 2.
* tempArray[1] = 7 (stored at 8), next address of tempArray[2] is 6. But 6 is smaller than address of currentPointer(means 8) => we have to read 01 at 0A => next address is 16
* Keep continue till tempArray is full of 18 integers.
* When encrypting password, we set specialArray[i] = per[pass[i]]
* Decrypting will find where specialArray[i] is in perArray and set it to pass[i]

For example:

PerArray:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 4 | 3 | 5 | 2 | 8 | 1 | 6 | 7 | 0 |

0 1 2 3 4 5 6 7 8 9

SpecialArray:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 7 | 3 | 2 | 9 | 9 | 6 | 8 |

First, we will find where is 3 (first element of SpecialArray) in PerArray => at PerArray[2] => first character of our password is 2

Keep continue, we have decrypted password is 28240075 => Correct

## IV. Work partition

|  |  |  |  |
| --- | --- | --- | --- |
|  | Anh Khoa | Cong Minh | Thanh Tu |
| Design File system | X |  |  |
| Code Boot sector |  |  | x |
| Code Cluster manager | X |  |  |
| Code create Vol | X |  |  |
| Code open Vol | X |  |  |
| Code import file | X |  |  |
| Code import folder | X |  |  |
| Code delete file |  |  | x |
| Code delete folder |  |  | x |
| Code explore file |  | x |  |
| Code explore folder |  | x |  |
| Code hash password |  | x |  |

## V. Menu

We have implemented Menu but because some mistake (bug) appear so we can’t complete this project. Very sorry