Jacob Kurian

11/22/2016

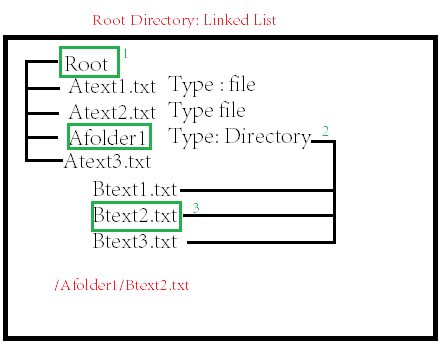
**Lab 4: File System implementation**

* General Data
* The size of the file system will be 5MB (5,243,000 bytes).
* The type of the file system is FAT 16.
* The file system has mainly four components: Reserved Sector (49bytes), FAT file (each cluster 20 bytes), Root Directory (92 bytes each cluster) and Data Section (32,772 bytes).
* All the numbers that I am writing to the file is in the form of a string[char array]. When I retrieve the those numbers I will convert the strings to numbers.
* The maximum size of a file is 32kb.
* The functions that available for the drive:
  + Create new file
  + Create a new Directory
  + Write to a file
  + Read from a file
  + Open file
  + Close file
  + Delete a file
  + Get information of a file (meta data and size).
  + Get general information of the disk. (total number of files, used space and available space)
* Also Drive implementation will handle with fragmentation issue.
* There is a defragmentation process right after a file deleted. So that we can utilize all the free space available in the data section.

**Overall Outlook of my File system:**

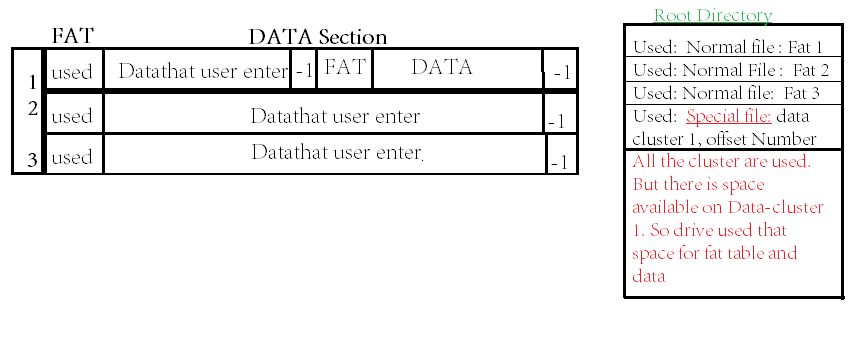
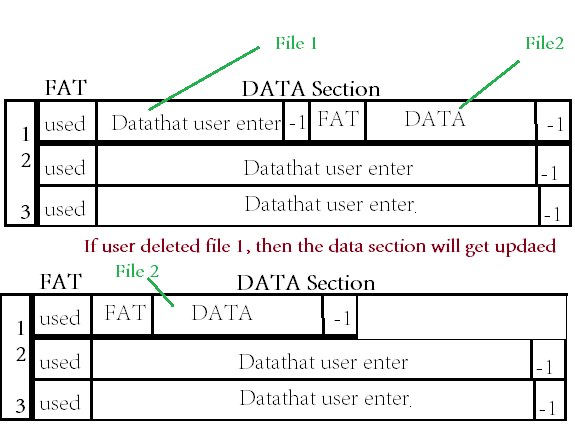
|  |  |  |  |
| --- | --- | --- | --- |
| Reserved Sector | Root Directory | Fat Table | Data Region |

**Documentation for Drive implementation**

* 'Reserved Sector' [49 bytes]
  + Information available on ‘Reserved Sector’
    - Number of total files in the drive . (0xfff = 5 bytes)
    - Remaining space in the drive (0xffffff = 8 bytes)
    - Used space in the drive(0xffffff= 8 bytes)
    - Bytes per cluster for data region [32,771 bytes](0xffff =6 bytes )
    - Bytes per cluster for FAT region(0xf = 3byte)
    - Bytes per cluster for root directory(0xff = 4 bytes)
    - Total number of bytes for Reserved Sector(0xff = 4 bytes)
    - Start of Root directory (0xffffff = 8 bytes )
    - Start of Fat table (0xff = 3 bytes)
  + Implementation: Reserved sector is the first sector of the drive. The program will get general information about the drive from ‘reserved sector’. This will implemented used basic ‘fwrite’ function. Everything will be implemented in an array.
* ‘Root Directory’ [92 bytes]
  + Information available on Root Director
    - Hex value to represent the cluster for root directory is available or not (oxf = 3 byte)
    - Length of the filename (Including the extension)(0xfff - 5 bytes)
    - Name of the file (16 bytes including the extension)
    - File type [text file or directory] (0xf = 3 byte)
    - File allocation type [oxf = 3 bytes] (If all fat table clusters are used, program will use the space available in data section for FAT table and store data. )
    - Created Date and time (25 bytes)
    - Modified Date and Time (25 bytes)
    - File size(0xfffffff= 8 bytes)
    - The start Cluster in FAT table (0xff = 4 bytes)
  + Implementation Details: It is implemented as linked list. If a file is a directory then it can have 2 pointers. First pointer will point to the sibiling file and next pointer will point to it’s children. Each file (non directory)will have all the components that mentioned above. Directories will have the created time and the total size.
* ‘Fat Table’ [20 bytes]
  + Information Available on FAT Table
* Pointers to the data section (0xff = 4 bytes [Total cluster number for data section may go upto 140-150])
* A byte to show either the cluster is used or unused (0xf = 3 byte)
* Offset in the Data cluster (0xffff = 6 bytes [have to go up to the number 32,768])
* Available space (0xfffffff= 8 bytes)
* Root directory number [useful to deal with defragmentation process 5 bytes]
* Implementation Details: The ‘FAT Table’ clusters have a Validity check to make sure whether the cluster is used or unused. If it is used, then the validity value will be 0x1 else it will have value 0x00. Also, it will have values that represent the available space in the Data cluster, the offset number, and the pointer to the ‘Data section’. Everything will be implemented in an array.
* Information available on ‘Data Region’ [32,772 bytes]
  + Information available on ‘Data Region’
* Data that user entered (32768 bytes = 32kb)
* A pointer to the next fat table next cluster if the file is more than 1 cluster long. Else it will say end of the file. (0xff = 4 bytes)
* Implementation Details: The user data will go here. There is a pointer at the end of the cluster to the next free block, if there is any. If the value of the pointer is -1 which means data ends in that cluster. The general idea for ‘Data section’ is linked list because ‘data section’ has pointer to next cluster if it is necessary. If there is no free cluster available in FAT TABLE , then program will use DATA section to store FAT.

**Documentation of Major functions:**

My drive has 4 major parts: Reserved sector, 1Fat Table, Root Directory, Data

* Create a file
  + Get the filename from the user. Go to the reserved sector and get the start cluster number of FAT table. Check for free FAT table cluster. If you find a free cluster, then mark it as used. Then update the pointer to ‘Data Region’, also update the size and offset of the cluster. Then go to the ‘Root Directory ’ and add the file to the linked list of the appropriate directory. Mark that cluster as used and put all the necessary info mentioned on page 1. Go and update the total number of files in ‘Reserved sector’.
* Create a directory
  + Go to the ‘Root Directory’, if user didn’t provide any specific path, then make the directory as the child of ‘Root Directory’. Else find the appropriate path and make the directory it’s path. If the path doesn’t exist, then show an invalid path message to the user.
* Open a file: Get the path name from the user. Go to the ‘Root Directory’ and traverse through the link and find the appropriate directory. Then compare the name of file that user provided with the file that exist in the directory. If you find the correct file, then get the FAT file cluster number. Go to the FAT file get the ‘Data section cluster number ‘ and offset. Then print the message to the screen that says “the file is open”. Also show menu option to the screen to\_read\_from\_file , write\_ to\_ file and close\_file. At this point, the program will enter in a while loop. It will only exit the loop if the user chooses ‘close\_file’ option. If the path or file name doesn’t exist. The program will print an “error message” to screen.
* How to write to drive: File should be already open.
  + Then go to FAT and get the offset number in the data section. And go to the data section. Check that the data will fit inside the space. If it does fit, then update the pointer at the end of the data section to -1. Else If, data don’t fit inside the space, then find the next free block . Update the pointer of data section to the next block. Do this until we write all the content or run out of free space. If we run out of free space, then look for fragmented spaces(call fragmente\_space\_management function that is described below). Go to ‘Reserved sector’ and update the values for available space and used space.
  + Fragment Management Function: If there are no unused blocks in the FAT. Go through all the Fat table cluster and look for available space greater than (32 + (size of allocation data bytes)/4)(20 bytes for FAT , pointerto next cluster[ 8 bytes for the offset and 4 bytes for the Data cluster number] ). Then go to the directory . Then mark the file as a special case because it's not going to the FAT table. Update the start cluster number in the ‘Data section’ and the offset number. Continue to write in the Data field until you finish writing your content or you run out of space in Data region.
* Read from the drive: If the file is already open then get the pointer to the ‘data section’ and offset number. Then go to “Data Section”read the contents from the ‘Data Section’. Store all the content in a buffer variable. Continue until the pointer in the ‘DATA section’ is -1. Then print the “buffer variable” to the screen. Then again show the menu options -> to\_read\_from\_file , write\_ to\_ file and close\_file.
* Close file: If the user chooses close file. The program will exit from while loop and go back to the main menu. Options that the main menu has -> create file, create directory, open file,delete file and eject .
* Delete File: Get the path and file name from the user. Go to the ‘Root Directory’ and find the file. Then mark the cluster that used for the file as unused. Also go to the ‘Fat Table’ mark the cluster as unused.
  + If the Data cluster store multiple files, then the defragmentation process will take place- > if there are fragmented data after the deleted file. It will move into the space of the deleted file. Update the available size in FAT file. Then go to the “Root directory ” of the moved file and update the details of start cluster number. The root directory number will be available in FAT table. Also mark the cluster of the ‘Root directory’ of the deleted file as unused.

Sudo Code:

char totalFiles[] = "0x000";  
char usedSpace[] = "0x000000";  
char remainingSpace[] = "0x00005";  
int static Reservedposition = 0;

int main(){

FILE \*fp = open(“filename.dat”, “r+”);

createReservedSector (fp);

while(Reservedposition < 5MB) {

createRootDirectory();

createFAT();

}

createDataSection;

do{

show option

1. createFile;
2. CreateFirectory;
3. Openfile;
4. Deletefile;
5. EjectDrive;

If(option selected == createFile){

Call the function: createfile();

Else if(option selected == createDirectory){

Call the function : createDirectory();

}

Else if(option selected == OpenFile){

Call the function: openFile();

}

Else if (selected option == Deleted file){

Call the function: deleteFile();

}

}

}while(!EjectDrive);

}

/\*Create the a file\*/

Void createfile(){

Char buff[100];

Printf(“Enter the file name: ”);

Scanf(“%s”, buff);

* Parse through the ‘buff’ variable
* Use strtok() make a double array that contains each word that user entered.

It’s just a file name.

Go to the ‘Fat Table’ and find the free cluster.

If(no free cluster found){

Call Special\_allocation\_funtion();

}

Else{

update the pointer to the corresponding ‘Data section’.

* Go to ‘Root Direcory’ find the correct directory and add the file as a new node.
* Then go to the FAT table and update the ‘Root directory’ number.

}

}

Void Special\_Allocation\_Function(){

* Go through each ‘FAT TABLE’ cluster and check for size greater than 32.
* Then make the fat file there.
* Go to ‘Root directory ’ and find a free block. Then update the information.
* Then go to the ‘Reserved Sector ’ find the correct directory and add the file as a new node.
* Then go to the FAT table and update the ‘Root directory’ number.

}

Void createDirectory(){

* Get the user path
* Parse the path name. Create a double array which contain all the words.
* Go to the ‘RootDirectory’ and find the appropriate path. Then add the new directory as the child of the directory located in array[length-2].

}

Void openFile(){

Char \*buff[100];

Print: “Enter the path name”

Scanf(“%s”, buff)

Char \*\*pathArray = pathParse(buff);

* go to the ‘Root Directory’ and traverse through the linked list and find the right path and locate the file.

do{

Print menu option

1. Read the file
2. Write to the file
3. Close the file

If(selectedOption == read the file){

Call the function: readFile();

}

Else if(selectd option == writefile){

Call the function: writefile();

}

}while(optionselected != close the file);

}

Void readfile(){

* Use the FAT table number from “root directory” to locate FAT Table cluster.
* Go to FAT TABLE, then get the pointer to the ‘DATA Section’.
* Go to ‘DATA SECTION’ and store all the data in a string variable(char []). Continue this until the pointer in the ‘DAT A’ section have -1 value.
* Then print the content of the string to the screen.

}

Void write(){

* Use the FAT table number from “root directory” to locate FAT Table cluster.
* Go to the FAT TABLE, then get the pointer to the ‘DATA SECTION’.
* Check the size of the content that user given fit in the available space of the ‘DATA SECTION’.

While(size of content written == size of content user provided || No free space){

* + If it does fit in the Data Section. Then write the content there.
  + Else, write the content upto the available free space. Then find the a new free block and write the rest there.
  + Cotinue until all the content have written or no more free space.

}

}

Void deletefile(){

Char \*buff[100];

Print: “Enter the file name”

Scanf(“%s”, buff)

Char \*\*pathArray = pathParse(buff);

* Locate the file in the “Root Directory”
* The change the used byte to unused.
* Get the’Fat Table’ cluster number. Go to ‘FAT TABLE’ cluster mark the cluster as unused.
* If multiple files allocated in the ‘DAT A SECTION’.
  + If there are fragmented data after the deleted file. It will move into the space of the deleted file. Update the available size in FAT file. Then go to the “Root directory ” of the moved file and update the details of start cluster number. The root directory number will be available in FAT table.

}

Char\*\* pathParse(char path[]){

* Make a double array using malloc function.
* That will have all the words in the path.
* The path will cut into different words using strtok function.
* Return ‘doubleArray’;

}

/\*Reserved section will have all the variables that mentioned in the ‘General Data ’ section (page 1)\*/

void createReservedSector(FILE \*fp,){

char bpcData[] = "0x8004"; //bytes per Cluster should change to bits per cluster[32,772 bytes]  
char  bpcDirect[] = "0x5c"; //bytes per cluster for ‘RootDirectory’ [92 bytes]  
char bpcFat[] = "0x014"; // bytes per cluster for FAT (20 bytes)  
char bpcReserved[] = "0x31"; // total number of bytes used Reserved Sector (49 bytes)  
char buff[100];  
int x;  
int readPosition = 0;  
  
  
fileWriter(fp, totalFiles); //Total Number of files in the drive  
fileWriter(fp, remainingSpace); // Total amout of remaining space  
fileWriter(fp, usedSpace); // Total amount of used space  
fileWriter(fp,bpcData); // Total space for data  
fileWriter(fp,bpcDirect); // Total bytes for directory  
fileWriter(fp, bpcFat); // total bytes for fat  
fileWriter(fp, bpcReserved); // total bytes for reserved sector

}

/\*Creating the FAT TABLE\*/

Void creatingFat(FILE \*fp){

Char pointToData [] = “0x00”;

Char validity[] = “0x0”;

Char offset[] = “0xffffff”;

Char availableSpace[] = “0xffffff”;

Char rootDirectoryNumber[] = “0xfff’;

fileWriter(fp, pointToData);

fileWriter(fp, validity);

fileWriter(fp, offset);

fileWriter(fp, availableSpace);

fileWriter(fp, rootDirectoryNumber);

}

/\*creating the Data section\*/

Void createDataSection(FILE\*fp){

Char dataSpace[] = “0x8000”;

Char ptrFat[] = “0x4”;

fileWriter(fp, dataSpace);

fileWriter(fp, ptrFat);

}

//Creating the ‘Root Directory’

Void createRootDirectory(FILE \*fp){

Char validity[] = “0x0”;

Char length[] = “0x000”;

Char filename[] = “0xffffffffffffff”;

Char filetype[] = “0xf”;

Char fileAllocationtype[] = “oxf”;

Char createdDate[] = “0xffffffffffffffffffff”;

Char modifiedDate[] = “0xffffffffffffffffffff”;

Char fileSize[] = “oxffffffff”;

Char startClusterFat[] = “0xff”;

fileWriter(fp, validity);

fileWriter(fp, length);

fileWriter(fp, fileAllocationtype);

fileWriter(fp, createdDate);

fileWriter(fp, modifiedDate);

fileWriter(fp, fileSize);

fileWriter(fp, startClusterFat);

}

// The function will write to the disk

void fileWriter(FILE \*fp, char str[]){  
char buff[6];

while(Reservedposition < 5MB){  
  fseek(fp, Reservedposition , SEEK\_SET);  
  fwrite(str, sizeof(char), strlen(str), fp);

Reservedposition = Reservedposition + strlen(str); // updating the ‘Reserved position’

}  
}