Amadeusz Deutry

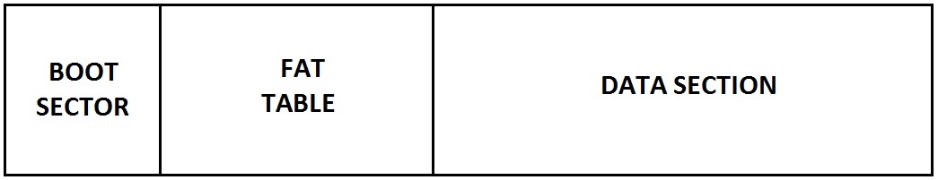
File System Documentation

**OVERVIEW**

A filesystem provides an operating system with the ability to read, write, and modify files on secondary storage devices. The responsibilities of an effective file system include the proper management of files and directories and minimizing the possibility of corrupt or damaged data. A file system should also provide the operating system with and user with a convenient and reliable interface which is able to perform operations on the directories and files already on the filesystem.

**DESIGN**

For my implementation of a filesystem I chose to follow the design of the FAT32 filesystem. The file system itself is stored in one file on a real machine. This file is separated into 3 parts: the boot sector, the FAT table, and the data section. (Traditional FAT filesystem implementations include two FAT tables for consistency but only one is included in this implementation.)



***Fig 1.*** *Overview of the filesystem structure*

The boot sector contains the size of the filesystem, the starting offset of the FAT table, the FAT table entry size, the starting offset of the data section and the data section cluster size. This is not included in traditional FAT system implementations but for this particular implementation it allows for various filesystem sizes.

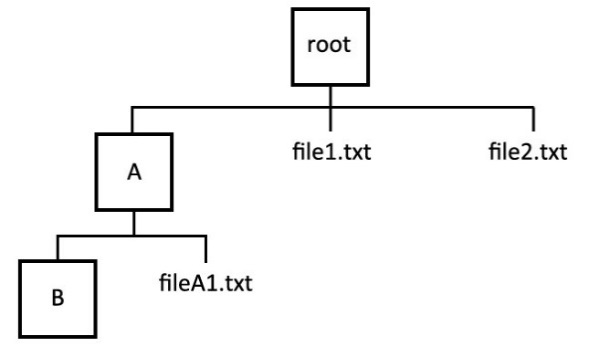
**THE FAT TABLE AND FILE ALLOCATION**

The disk on which the filesystem resides is divided into equally sized portions called clusters. The FAT table is a list of entries each of which map into a cluster which resides on the physical disk. In this implementation the physical disk is separated into clusters of 512 bytes and the FAT table entries are 4 bytes each. Of course files can be bigger than the space provided by a single cluster. Therefore clusters can be chained together into a linked list allowing files to make use of several clusters. This is achieved by including a pointer to the next cluster within each FAT entry. Every file has an associated beginning FAT entry which maps into the very first cluster that the file occupies. When the file system needs to retrieve the file it first reads all the data from the first cluster and then uses the pointer inside the FAT entry to locate the next cluster. A special marker is set to specify that a given cluster is the last cluster of the file. The first entry in the FAT table is reserved for the root directory which resides in the data section.

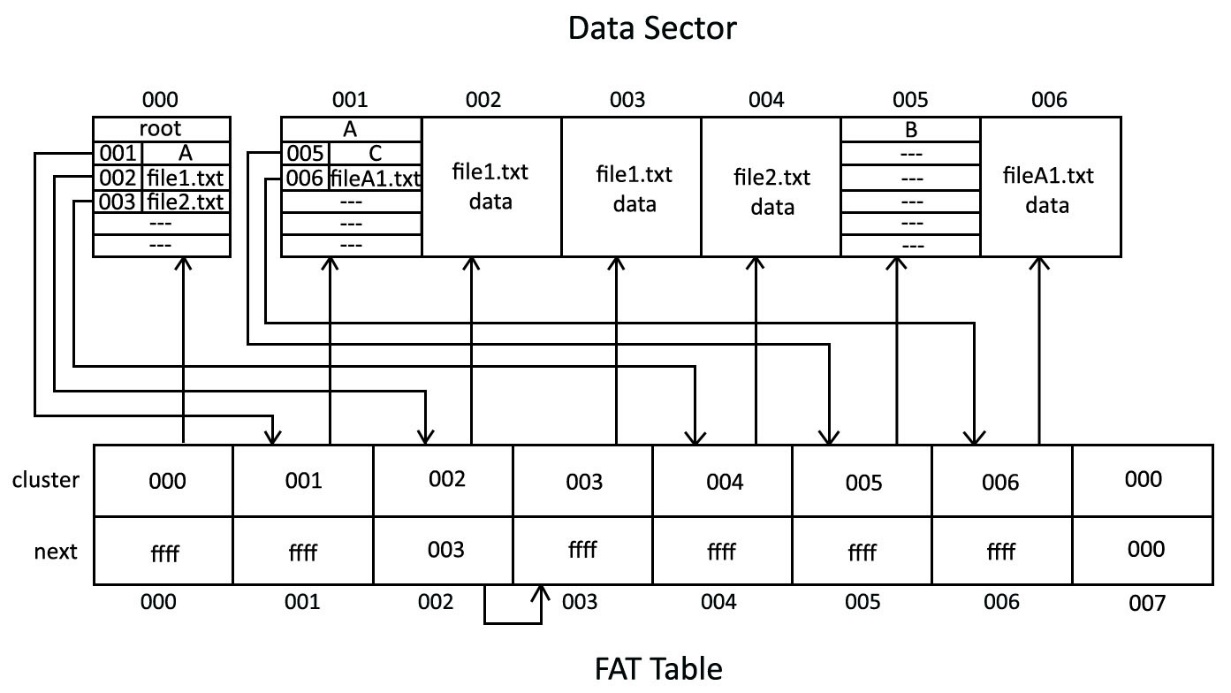
**THE DATA SECTION AND PHYSICAL ALLOCATION**

The data section includes the actual file data including the directory tables. As mentioned before the data section is divided into 512 byte clusters. The data section includes directory tables which are special files that describe the contents of a filesystem directory. A directory table includes entries for each item in the directory. Each entry is 32 bytes in size and includes information such as whether the item is a file or a directory, a pointer to the first FAT cluster of the item, the file name, and dates of creation and last modification. Entries can specify whether the item is a directory or a file. This allows for directories to contains subdirectories.

The two diagrams below describes a simple directory file structure and its representation in the FAT table and data sector. Each box represents a directory with the children drawn underneath, connected to the parent. In this diagram the root directory contains two text files and a directory named A. The A directory contains another text file and its own subdirectory named B which happens to be empty.



***Fig 2.*** *An example directory structure*



***Fig 3.*** *The example directory structure as stored in the FAT table and data sector.*

The first entry in the FAT table is reserved for the root directory therefore it also points to the first cluster in the data sector. The root directory table contains an entry for each of the directory’s three items. In the diagram we can see that each entry contains the cluster number of the item’s first cluster in the FAT table and the item’s name. In the actual implementation this directory table entry contains additional information including the item type ( file or directory ), the date and time of creation and last modification and the file size. The cluster number maps into a FAT table entry which is shown as the bottom table in Fig 3.

A FAT entry includes the cluster number of the next cluster in the file chain and a pointer to the cluster which this FAT entry maps to. The purpose of including a data field which specified the next cluster in the file chain is to give the filesystem the ability to store files which can occupy multiple clusters. For example the file file1.txt contained in the root directory cannot fit into a single cluster and requires the space of two clusters. We can see that this file’s first FAT entry contains a next value of 003 which points to the FAT entry of the next cluster. The next value of this FAT entry is “ffff” which specifies that this is the last cluster in the cluster chain. From the example can see than every allocated file or directory besides file1.txt can be stored inside a single cluster because the value of “next” is “ffff” for each one.

A filesystem must of course be able to differentiate between files and directories when traversing its directory structure. An important detail to note is that the directory tables allow the filesystem to differentiate between files and directories by storing a value in each directory table entry which specifies the entry as being either a file or directory. Therefore the way that the filesystem treats each data cluster depends on the context which is provided by the data table entries.

Running through an example of a simple file retrieval can provide us with a better understanding of how the filesystem structure operates. Let us suppose that the file system receives the command to open the file located at “/A/fileA1.txt”. We can assume that every file or directory path will be given as an absolute path (relative to the root directory). The first step that the filesystem must perform is locate entry ‘A’ in the in the root directory. The filesystem reads the very first FAT entry in the FAT table which contains a mapping into the first cluster of the root directory. After reading the first cluster it checks the ‘next’ value of the FAT entry to see if the root directory occupies any more clusters. If it does not it passes the data it retrieved from the cluster into a function that handles directory table data. The function parses the directory table data and retrieves the information about each entry including the name and type of the entry. Because we requested to open a file inside the ‘A’ entry of the root directory we must make sure that the ‘A’ entry exists and that it is also a directory. If either of these are false we return the appropriate error. After we have validated that the ‘A’ entry exists and is indeed a directory we can use the entry’s FAT pointer number to locate the first FAT entry of the directory. In this example the first FAT entry of directory A is located at cluster number 001. We can retrieve the FAT entry of this directory by taking the starting index of the FAT table and adding to it the cluster number of the entry times the size of each FAT entry. By looking at FAT entry 001 we can see that directory ‘A’ occupies only one cluster because the next value of its first cluster is ‘ffff’. We load the cluster data and treat it as a directory table just as we did with the root directory and extract the directory information. Our next step is to make sure the file named ‘fileA1.txt’ exists inside the directory and if it doesn’t we throw an error. If the file does exist we navigate to its first FAT entry given by the cluster number in the directory table entry and begin to retrieve the data continuing to navigate to the next cluster in the chain until the final chain has been reached. In this case, however, the file ‘fileA1.txt’ occupies only a single cluster so we do not need to navigate to any other FAT entry.

**IMPLEMENTATION**

The filesystem is stored on a single file which contains the boot sector, FAT table, and data section. This data must first be written to a file filesystem initialization procedure called fs\_init. The fs\_init function takes as arguments a system file pointer,a boot sector size, a FAT entry size, the desired FAT table length, the size of data clusters and the size of the data section. All of the specified sizes are given in bytes. The function assumes that the file was opened in binary mode instead of text mode. The function creates a data struct which will contain all of these values and writes this struct to the beginning of the filesystem file. The function must also initialize the root directory. Directories in this implementation include one small struct which stores the amount of items in the directory,

|  |
| --- |
| int fs\_init(FILE \* fs\_file,  unsigned int boot\_sec\_size ,  unsigned int fat\_entry\_size ,  unsigned int fat\_table\_size,  unsigned int cluster\_size,  unsigned int data\_sec\_size){  /create a boot sector struct  struct boot\_sec init\_boot\_sec;  //assign values to the struct  init\_boot\_sec.boot\_sec\_size = boot\_sec\_size;  init\_boot\_sec.fs\_size = boot\_sec\_size + fat\_table\_size + data\_sec\_size;  init\_boot\_sec.fat\_table\_size = fat\_table\_size;  init\_boot\_sec.cluste\_size = cluster\_size;  init\_boot\_sec.data\_sec\_size = data\_sec\_size;  init\_boot\_sec.root\_start = boot\_sec\_size + fat\_table\_size + 1;      //point the filesystem file pointer to the beginning of the file  fseek(fs\_file , 0 , SEEK\_SET);  //write the struct to the filesystem file  Fwrite(init\_boot\_sec)  //initialize the directory struct  Struct directory root;  Root.item\_count = 0;    //write the directory struct in the index following the fat\_table  Fseek(fs\_file , boot\_sec\_size + fat\_table\_size + 1 , SEEK\_SET);    } |

Now when we wish to make use of the filesystem the filesystem will have to read the boot sector struct from the beginning of the file in order to retrieve the indexes of the fat table and root directory.

|  |
| --- |
| Struct boot\_sec fs\_info;  Fread(fs\_info , sizeof(boot\_sec) , fs\_file) |

The fs\_info struct can now be used to retrieve the various filesystem information.

**OPEN FILE**

Open\_file takes an absolute path name as a parameter and returns a file system file descriptor. The file system descriptor is a wrapper around the operating system file descriptor which is used by the file system to properly retrieve data from clusters. The function returns NULL if the file does not exist.

|  |
| --- |
| File\_descriptor open\_file(char\* path)  {  Char\*\* args = parse\_path(path)  Struct directory cur\_dir;  Struct dir\_table\_entry cur\_entry;  //we start at the root directory  Cur\_dir = get\_root\_dir  Entries = get\_dir\_entries(cur\_dir)  I = 0    For each arg in args:  For each entry in entries:  If entry == args[i]  if I == length(path)  Struct file\_descriptor fd;  Fd.root\_cluster = entry.cluster  return fd  else  cur\_dir = entry  break  i++      } |

**CREATING FILES**

Creating a file involves retrieving the file descriptor for the directory in which we wish to create the file, adding a directory table entry for the file, and allocating the necessary amount of FAT entries and data clusters to fit the file. Create\_file takes in the path of the desired file as an argument which includes the name of the file and its extension at the end.

|  |
| --- |
| File\_descriptor create\_file(String path)  {    } |

**CLOSING FILES**

|  |
| --- |
|  |

**WRITING FILES**

|  |
| --- |
| void write\_file(void\* buffer , fs , int index, int bytes){  } |

**READING FILES**

|  |
| --- |
| Void read\_file(void\* buffer, file\_descriptor fs, int index, int bytes)  {  Struct fat\_entry cur\_fat;  Struct dir\_table\_entry entry;  //read the directory table entry from the given file descriptor  Fread(entry , root\_start + fs.root\_cluster\*sizeof(dir\_table\_entry))  //retrieve the first FAT sector from the directory table entry  Fread(cur\_fat, boot\_sec\_size + entry.fat\_index\*sizeof(fat\_entry))  //navigate through the file chain until you’ve reached the index  While(index > 512)  Entry = \*(entry.next)  Index -= 512  //read bytes number of bytes from the appropriate sector  Fread(buffer ,root\_start + entry.cluster\*sizeof(cluster) + index, bytes)    } |

**DELETING FILES**

The delete\_file function will take a file descriptor as an argument, delete the file entry to which it points, and free up the FAT entries that the file occupies.

|  |
| --- |
| Int delete\_file(file\_descriptor fd){  Struct dir\_table\_entry file\_entry = malloc    Fread(file\_entry, root\_start + fd.cluster \* cluster\_size)    //free up the FAT entries associated with this file  Unallocate\_blocks(file\_entry.fat\_index);  //delete the directory entry data  zero(fd.cluster , 32)  } |

**FUNCTIONS**

|  |
| --- |
| //the boot sector struct  struct boot\_sec{  unsigned int boot\_sec\_size:16;  unsigned int fs\_size:32;  unsigned int fat\_table\_size:32;  unsigned int fat\_entry\_size:8;  unsigned int cluster\_size:8;  unsigned int data\_sec\_size: 32;  unsigned int root\_start: 32;    };  Struct fat\_entry {  Unsigned int cluster:16;  Unsigned int next:16;  }  Struct directory {  Unsigned int fat\_index:8;  Unsigned int item\_count:8;  }  Struct dir\_table\_entry {  Unsigned int fat\_index:8;  Unsigned int name:12;  Unsigned int size:16;  Unsigned int type: 1;  }  Struct file\_descriptor {  //the cluster where the file resides  Unsigned int root\_cluster:16;  //the entry number of the file inside the directory  Unsigned int entry\_no:16;  }  Struct Dir\_table\_entry\* get\_dir\_entries(directory dir)  {  Ret = malloc(dir.item\_count)  For each entry in dir:  Struct Dir\_table\_entry temp = entry  Ret[i] = entry  Return ret    }  Directory get\_root\_dir()  {  Struct directory temp = malloc  Fread(temp , root\_start , fs\_file)  Return temp  }  //TODO  Directory open\_dir(String dir\_path)  {  Struct directory ret = malloc  Struct file\_descriptor fd = open\_file(dir\_path)  Fseek(fs\_file, ,SEEK\_SET)  Fread(  }  //this function unallocates all the blocks in a FAT entry chain by iterating  //through each of the individual FAT entries and setting their next values  // to null and the allocated value to 0  void unallocate\_blocks(fat\_entry entry)  {  while(entry.next!=NULL)  {  entry.next = NULL;  entry.allocated = 0;  entry = entry.next;  }    //clear the last FAT entry  entry.next = NULL;  entry.allocated = 0;    } |