Kyle O’Neill

3207 Operating Systems

File System Design

3 April 2016

File System Design

**Introduction**

The goal of this project is to create a “disk” that will store data from files and organize them by directories. This is generally how data is stored on any type of storage on a computer. The file system that I will be modeling my disk after is FAT16. FAT stands for File Allocation Table. The file table holds pointers to the blocks of memory, which is where all the file data is stored. This table is set up as a linked list and any blocks that are associated with the same file will be linked together.

Each FAT entry will contain 3 pieces of data. Those three pieces are:

1. Block Number
2. Next Entry
3. Previous Entry

All of this data will allow my file system to determine which block is associated with each file. The block number will be the pointer to the block of data. The next and previous entries are going to contain the links between the file table entries. This links will allow a file to take up more than one block of data. The FAT entries will each be 4B long. These values will also be able to tell the program if that block is free or not.

Below the FAT is the data blocks. These blocks is where the data inside all the files is located. A block is a way to divided the data between all the files on the system. Each of these blocks are only associated with a single file. In my file system, each block is going to hold 512B of data. The data blocks are going to be organized according to the block numbers going from 0 to Number of Blocks - 1. These FAT blocks are going to be the overhead needed to organize the disk.

The file system is going to organize files into directories so the user can divide the files however they choose. To represent a directory, the file system is going to fill the data blocks that it is associated with pointers to all the files and other directories that are located inside it. In doing this we treat the file exactly like files which makes implementing the file system easier.

Specs of File System:

* Disk Size: 2MB
* Block Size: 512B
* Number of Blocks: 4000
* Max File Size: 32KB
* FAT Table Entries: 4B
* File Header: 16B

Structure of File System

---FAT

| F ----

| F ----

| F ----

…

| F ----

---Blocks

| B ---------------------------------

| B ---------------------------------

| B ---------------------------------

…

| B ---------------------------------

**Definitions**

Block:

A single section of the disk which is 512B long and contains the data inside each file

File Header:

Data that is stored inside a block which points to the data inside the file and some metadata about the file.

FAT Entry:

A pointer to a block of data that is contained inside the file system. Also contains links to other block associated with the same data.

“Disk”:

The file that will be used to simulate a hard drive.

File:

A collection of bytes that are associated with one another.

Directory:

A collection of files and other directories.

**File Interface**

The interface of my file system will be comprised of seven different functions, each using the built in system calls in the C library. These functions will allow me to access, save and transverse the file system to perform the necessary tasks required.

My file system is going to use a FAT table to represent each memory block located inside the disk. Inside each entry, is going to be indicator determining what information is located inside the block. These indicators will be located inside the FAT

There are three different types of information that can be stored inside a table entry. They include a file, directory or a free block. To represent a free block. The value of both the previous and next links inside the FAT table is going to be 0xFFFF. That number exceeds the total number of block so it will be used to represent a free block. A similar technique will be used to represent a directory. The previous link in the FAT table will be 0xFFFE. This value also exceeds the total number of blocks in the file system. Files will be represented by table entries without those two values in one of their links.

To read and write data from my file system, the interface will determine which blocks or set of blocks that are needed. Once those blocks are found, the program will transverse the file until it finds the block it is looking for. It will then perform the task which the interface is requesting. (read, write, etc.) If it requires more than one block, the interface will use the FAT table to determine and create links between each of the file blocks. This allows files or directories that require more data storage to use multiple blocks. This is a general picture of how the file system is going to operate.

**Functions Pseudocode**

Open File – koOpen

* Open disk with system call
* Parse out and store FAT table into an array
* Get directory file is in which is passed as function arg
* Iterate through directory blocks until file directory is found
  + If not fount return file not found
* Iterate through the file directory block, searching for file
  + If not found return file not found
  + If it is found update time stamp
* Find first block associated with file
* Return open file with pointer starting at the new file

Create File – koCreateFile

* Open disk with system call
* Parse out and store FAT table into an array
* Get the location of the file to be inserted
* Iterate through directory blocks to access the directory block associated with that directory
  + If directory is not found return directory not found
* Find the next free space inside the directory block that can hold the file header.
* Search through FAT Table to find next free block
* Place file header into the block with a pointer to the FAT table entry associated that file.
* Update FAT table and rewrite over the FAT Table located on the disk.
* Return Success

Create Directory - koCreateDir

* Open disk with system call
* Parse out and store FAT table into an array
* Get the location of the file to be inserted
* Iterate through directory blocks to access the directory block associated with that directory
  + If directory is not found return directory not found
* Find the next free space inside the directory block that can hold the file header.
* Search through FAT Table to find next free block
* Place directory header into the block with a pointer to the FAT table entry associated that file.
* Update FAT Table and rewrite over the FAT Table on the disk
* Return Success

Close File – koClose

* Rewrite the FAT Table stored in the array onto the disk
* Perform system close file code

Write File - koWrite

* \*File must be open using koOpen\*
* Check to make sure the disk pointer is in right place
  + If it is at the end of a block
    - Find next free block
    - Create a link from the previous block
    - Set disk pointer to beginning of new block
    - Update FAT Table and rewrite to the disk
* Write the byte where the file pointer is at.
* Return success

Read File - koRead

* \*File must be opened using koOpen\*
* Check to make sure the disk pointer is in right place
  + If it is at the end of a block
    - Find the next block associated with the file in the FAT
    - Set disk pointer to beginning of new block
* Return’s the byte where the file pointer is at.

Delete File - koRemoveFile

* Open disk using system call
* Get FAT table from the file and store into the array
* Iterate through directory pointers to find the directory that the file is located in
  + If the directory doesn’t exist, return file not found
* Go to file block associated with the file to be deleted.
* Loop through all the bytes in the block and fill in with 0’s
  + Use a counter to determine when the end of the block is reached
  + If the end of the block is reached, find link to next block
    - Go to that block
    - Reset counter
* Update FAT Table and rewrite to disk
* Return success
* Reset disk pointer to root directory