**Writeup:**

**Comments:**

**Code:**

**----------------------------------------------------**

**CS 4414 Operating Systems – Fall 2017**

**Homework 3: FAT Read Library**

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly for both FAT16 and FAT32.

1. **Problem Description**

The objective of this assignment was to become familiar with file system organization and learn how file allocation tables (FATs) are used to organize disk space. The program had to provide functions to change directory, list entries of a directory, and open, close, and read files contained in a FAT16 or FAT32 formatted volume.

1. **Approach**

In order to use the library created for this assignment, a FAT formatted volume must be designated by setting the environment variable FAT\_FS\_PATH. There are five library methods that are implemented in this assignment. *OS\_cd* changes the current working directory. *OS\_open* locates a file at a specified path and returns a file descriptor that can be used to read the file. *OS\_close* closes a file specified by a file descriptor. *OS\_read* reads a set number of bytes from a given file into a buffer. *OS\_readDir* returns a list of entries in a specified directory. The library also defines a structure called *dirEnt* which represents a directory entry as described in the FAT specification. The current working directory is represented by a pointer to a list of *dirEnt* structures representing entries in the current working directory.

At the beginning of each method, the program checks if the FAT volume has been opened for reading. If it has not been opened, the volume is opened and basic information is recorded using the *init\_fat* method. The purpose of *init\_fat* is to load basic information about the FAT volume, store that information for later use, and initialize the current working directory and other critical information. The first 36 bytes of the FAT volume are read into a *BPB\_structure* which represents the BIOS Parameter Block (BPB) as described in the FAT specification. The program reads the byte sequence starting at the end of the BPB into extended boot record (EBR) structures for both FAT16 and FAT32 systems. The information contained in the BPB and EBR is used to determine the count of clusters in the system. From this count, the file system type, FAT16 or FAT32, is determined and stored in a global variable. Next, the root entries are read in. For FAT16 volumes, the location of the root directory can be calculated from the BPB and the number of root directory entries are specified by the BPB by the field *BPB\_RootEntCnt*. The system starts at the root sector and reads information in *BPB\_RootEntCnt* consecutive *dirEnt* structures. For FAT32 volumes, the cluster containing the root directory entry is stored in the FAT32 EBR, and the entries are read in using the *read\_cluster\_dirEnt* method. A pointer representing the list of these root directory entries is stored globally, and the current working directory pointer is set to this pointer.

The *read\_cluster\_dirEnt* method is one of several helper methods that are used in this library. The purpose of this method is to read a list of directory entries from a given cluster. First, the length of the cluster chain is calculated by following the entries in the FAT table starting at the given cluster. This is done by calling the *value\_in\_FAT* method, which retrieves a FAT table entry at a specified cluster using the process in the FAT specification, repeatedly until the “no next cluster” value (y > 0xFFF8 or y > 0x0FFFFFF8 for FAT16 and FAT32 respectively) is found. The total number of clusters in the chain multiplied by the number of directory entries that can be contained in a single cluster and a pointer containing that number of *dirEnt* structs is allocated. The function then reads 32 bytes at a time, starting from the sector corresponding to the first cluster, into the *dirEnts* represented by this pointer. When the end of the cluster is reached, the next cluster is found by consulting the FAT table. This process is repeated until either the end of the final cluster is reached, or a *dirEnt* is read with a first byte value of 0. This indicates that there are no more entries to be read, and the method returns the pointer containing the directory entries.

The *findDir* and *findDirEntry* methods are also useful in implementing the five library functions. The *findDirEntry* method finds a directory entry in a list of directory entries matching a specified name and stores it in a pointer passed as an argument to the function. The pointer of directory entries is iterated through until either an entry with a first byte of 0 is found, indicating no more entries, or a name match is found. The names are matched either by using a sequence of long filename structures or by using the 8.3 specification. This program does support long filenames, but because they are beyond the scope of this assignment, the implementation will not be discussed further here. The 8.3 specification creates a filename from an 11 character directory entry name by placing a period in between the first 8 and last 3 characters. This, “CONGRATSTXT” becomes “CONGRATS.TXT”. When a directory entry is found with a name matching that specified by the argument to *findDirEntry*, it is stored in the pointer passed to the function and a value of 1 is returned. If no matching entry is found, the pointer is left unmodified and the return code is 0.

The *findDirEntry* function is used to implement the *findDir* method, which is the most important helper method for this library. The purpose of *findDir* is to return a pointer to a list of directory entries contained at a specified path, given a directory to start from. The starting directory is represented by a list of directory entries contained in that directory, similar to how the current working directory is represented. This function separates the path name into *first* and *remaining*, where *first* is the first entry in the path and *remaining* is the path to be followed after the first entry. The function then calls *findDirEntry* on the current directory to match an entry to the name in *first*. If no such entry is found, the function returns NULL. Otherwise, if that directory entry has an attribute corresponding to a directory, the list of entries in that directory is retrieved using *read\_cluster\_dirEnt* and stored in *next\_dir*. Next, *findDir* is called recursively on this list of entries with the new path given by *remaining*. The base case for the recursive call is when the specified path is empty, upon which the function returns NULL. If an upper level function receives NULL from a recursive call to *findDir*, it will return *next\_dir* if it sees *remaining* is empty and will return NULL otherwise. If the upper level function receives a result that is not NULL, it will return that result. At the top level, the function returns the list of directory entries in the directory at the location of the specified path.

With the described helper methods, all of the library functions can be implemented simply. *OS\_readDir* is a wrapper around *findDir* to return a list of directory entries contained at a relative or absolute path. If the specified directory name contains a leading slash, *findDir* is called with the stored root directory entries from *init\_fat* as the current directory. Otherwise, *findDir* is called with the current working directory. In either case, the results of *findDir* are returned by *OS\_readDir*.

*OS\_cd* accepts a path and sets the current working directory to that path. This is done by calling *OS\_readDir* and setting the list of directory entries representing the current directory to the results of *OS\_readDir*. If these results are NULL, the function returns -1 and the current directory is left unchanged. Otherwise, it is modified and the function returns 1.

*OS\_open* accepts a path to a file, generates a file descriptor for the file, and returns it. The file is located by separating the path into the filename, which is the section of the path string after the final slash, and the file path, which is the part preceding the final slash. The function then calls *OS\_readDir* to get the directory entries at the location specified by the file path. Next, *findDirEntry* is called on this list of directory entries with the filename as the name to be matched. If either of these methods fail, the function returns -1. Otherwise, a file descriptor for the file is found and returned.

File descriptors in this program are represented by a large array of integers and a corresponding array of *dirEnt* structures. The integer array stores the first cluster number of the file pointed to by each file descriptor, or -1 if the file is unopened. The *dirEnt* array stores the *dirEnts* representing the opened files. This array is stored to ensure that no reads are performed past the end of a file. The integer array was created separate from the *dirEnt* array in order to allow for future implementation of a *seek* method, which would allow a file to be read at a specified offset. In order to find a file descriptor, the integer array is iterated through until an entry with value -1 is found. The index of this entry is the file descriptor, and the first cluster of the file and its *dirEnt* are stored in the two arrays.

When *OS\_close* is called, the entry in the integer array of file descriptors corresponding to the argument is set to -1. If this entry is already set to -1, or if it is outside the bounds of the array, the function returns -1. Otherwise, the function returns 1.

The final method in the library is *OS\_read*. This function allows a user to read a number of bytes into a buffer from an opened file starting from some offset. If the file descriptor passed as an argument does not correspond to an opened file as per the file descriptor arrays, the function returns -1. Otherwise the function attempts to read the specified number of bytes into the given buffer. There are a few complications that can arise that *OS\_read* seeks to handle. First, it is possible that the specified offset goes past the end of the file. To handle this, the function checks whether the offset is either greater than the file size as specified by the file’s *dirEnt*, or if the offset would result in reading at a cluster past the end of the file. The second condition is evaluated by counting the number of clusters the file spans by following the cluster chain in the FAT, and checking if the offset, converted from bytes to clusters, is greater than this number. If the offset is valid, the function reads the file until the desired number of bytes has been read, or until the end of the file, whichever comes first. If the number of bytes to read will result in reaching the end of a cluster, the function reads until the end of the cluster, seeks to the next cluster, and starts reading again from there. Once the desired number of bytes has been read, or the end of the file has been reached (which happens when the number of bytes read is equal to the file size specified by the *dirEnt* for the given file descriptor), the function returns the total number of bytes read from the file.

1. **Problems Encountered**

One significant problem encountered was posed by the “.. “ entries for subdirectories of root. In all other directories, this entry has a first cluster number corresponding to the directory file of the current directory’s parent. However, for those directories whose parents are root, this entry had a cluster number of 0 in the FAT volumes given. This would result in an incorrect sector to read from, causing calls such as *OS\_cd(“/People/../”);* to fail. To circumvent this problem, *read\_cluster\_dirEnt* was modified to check if the cluster argument was 0. If so, the function started reading from the calculated root sector from the BPB in FAT16, or from the root cluster specified by the EBR in FAT32.

Another set of problems were those posed by *OS\_read*. It is possible for users to specify an offset past the end of a file, which could result in reads to other files. In addition, it is possible for the desired number of bytes to be more than the number contained in the opened file. These problems were resolved by checking the offset before beginning to read, and by counting the number of bytes read and terminating the read when this number reaches the file size.

1. **Testing**

The library functions were tested on two given FAT volumes. One was formatted using FAT16, and the other was formatted using FAT32. In the testing program, the *OS\_cd* and *OS\_readDir* functions were called on a variety of path names. These path names could be relative or absolute, valid or invalid, and often contained several “..” entries to check the problem described earlier. The structure of the given volumes were inspected using the *hexcurse* hex editor, and the output of *OS\_readDir* was compared to the raw data in this volume. The byte offset in the volume was tracked through extensive print statements at first to ensure that the programs were navigating the volume correctly. The correctness of *OS\_cd* was evaluated by ensuring valid output from *OS\_readDir* called on relative pathnames after *OS\_cd* calls.

*OS\_open* was called on a variety of real and fake filenames to ensure that file descriptors were only being returned when appropriate. *OS\_close* was called on valid and invalid file descriptors to ensure that only opened files could be closed. *OS\_read* was called on several files such as “CONGRATS.TXT” in root, “SOARING.TXT” in some of the subdirectories of “/PEOPLE/”, and several of the photos in “/MEDIA/”. These files were read at various byte offsets and for various numbers of bytes in order to check that users could not read past the end of a file and that cluster transitions were also handled correctly.

The structure of the FAT16 and FAT32 volumes were the same. In order to test the FAT32 functions, the FAT16 functions were first checked for correctness as described above on the FAT16 volume. Next, the testing script was run on both the FAT16 and FAT32 volumes, and the outputs for each volume were compared with the *diff* command. When *diff* returned no difference between the two outputs, the FAT32 functionality was determined to be correct.

1. **Conclusion**

This assignment was successful in teaching file system structure and the use of file allocation tables in keeping track of disk space. I have learned how to use the various structures in the FAT specification to allow for navigation through the raw data of a filesystem and reading on files contained in that filesystem. The required library functions for both FAT16 and FAT32 work as expected and run quickly.

1. **Pledge**

On my honor as a student I have neither given nor received aid on this assignment.