Write up
Comments
Code

Operating Systems Homework #3

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to create a read-only API on a FAT16 file system. Given a FAT16 file system, the library functions need to navigate the allocation table and the clusters within the system, performing the appropriate read operations. 5 functions were implemented:

- *OS_cd(const char *path)* – changes the current working directory to the specified path
- *OS_open(const char *path)* – opens a file descriptor to the file specified by the path
- *OS_close(int fd)* – closes the given file descriptor
- *OS_read(int fildes, void *buffer, int nbyte, int offset)* – given a file descriptor to a previously opened file, reads nbytes into the given buffer from a given offset into the file
- *OS_readDir(const char *dirname)* – reads the directory entries at the specified directory

2. Approach:

Following the FAT16 specification document to navigate around the file system was crucial to the success of this assignment. First, structs were defined to extract sections of the file system in a manner that would be convenient to access particular fields. The following structs were defined:

- **fat_BS_t** – holds the boot sector located in the beginning reserved sectors of the file system, containing specification information about the system
- **dirEnt** – defines the structure of each directory entry, which contains information about the associated directory or file

For all API functions, it was important to first read in the boot sector into memory in order to have readily accessible information about the file system. The FAT system specification document contains various useful calculations to determine important information such as where the data sector starts, where a specific cluster starts, reading entries in the FAT itself, etc. Functions were implemented to calculate these values so that they could be invoked where necessary:

- *getFirstRootDirSecNum()*
- *getFirstDataSector()*
- *getFirstSectorOfCluster()*
- *getFatValue()*

- *getBytesPerCluster()*

For the library functions in general, some state needed to persist across function calls. In particular, the current working directory needed to be stored, as well as a local file descriptor table to keep track of open files. These were implemented as static variables in the library, with a char* string to keep track of the current working directory and an array of unsigned shorts to maintain the beginning cluster numbers of open files pointed to by file descriptors.

With this information setup, each library function can be implemented to perform the appropriate read operations. A recurring operation that occurs in multiple API functions is following a file path and traversing down multiple directories. For example in *OS_cd()*, the file path needs to be traversed to ensure that all parts of the file path are valid directories. In *OS_open()*, the filepath needs to be traversed to find the specified *dirEntry*. In *OS_readDir()*, the file path also needs to be traversed to locate the list of *dirEntry*'s in the specified directory. This method was implemented as *traverseDirectories()*, which splits the filepath into parts using the *strtok()* function and scanning through lists of directory entries to find ones that match parts of the file path. When an appropriate directory entry is found, it is verified to be a directory (not a file) and the cluster number is followed to find the next list of directory entries in the data volume. This eventually returns a real file descriptor pointing to the beginning of the data volume containing the last part of the file path. For instance, if "*/home/student/foo.py*" is traversed, a file descriptor pointing to the beginning of the *student* volume for "*/home/student"* is found. The appropriate library function will differ in operation after this.

The other function of interest not mentioned yet is *OS_read(),* where following the cluster chain is more important in order to read large files that span across multiple clusters. This function reads the FAT table to find the next cluster in the chain, or to determine it has reached the end of the cluster. Calculations to find the FAT entry for a specific cluster are implemented in *getFatValue()* listed above.

Another interesting discussion point is the handling of absolute vs. relative file path names. To handle this, all relative file paths are converted to absolute file paths, and the *traverseDirectories()* function traverses appropriately down the full absolute file path.

3. Problems Encountered

One major difficulty encountered was compiling the program into a shared library and linking it to another program to test the library functions. Multiple sources were consulted, as well as the TA, to figure out how to properly link the shared library into a test program.

A few other challenges were encountered when implementing the library functions itself. The hardest parts involved implementing the proper calculations to access particular parts of the file system, as specified in the FAT document. Issues were encountered in terms of the unsigned types and casting values for appropriate calculation. Many of these issues were fixed by looking at the file system through a hex editor to aid in debugging.

Another major challenge was implementing the *OS_read()* function that involved reading values into the buffer. It took some pointer arithmetic to determine how to continually read to the same buffer for files that spanned multiple clusters. Data had to be read into the buffer up until the end of the cluster, and then the file descriptor had to be shifted to the next cluster to resume reading into the buffer from where it had left of. Pointer arithmetic was performed to update the location in the buffer to read values into, as well as calculations involving the number of bytes remaining to be read and the number of bytes in a cluster.

4. Testing

To test these library functions, a test program was written that linked the shared library, and each function was extensively tested. The *OS_cd()* function was tested repeatedly throughout the test program, which changed the directory multiple times to test relative file path arguments for the other functions. The *OS_open()* function was tested in conjunction with the *OS_read()* function by opening file descriptors first and then reading from them. Reading was tested under multiple conditions. Text files were read (files under the PEOPLE directory) and printed to verify correctness. Text files were also read with the offset invoked, to ensure that the beginning characters that were offset were missing. However, most of these text files only spanned a single cluster. To test that the FAT entry reading and cluster chain following worked, the larger MEDIA files were tested on. This was tested by reading an image or movie file in the FAT16 file system, and these bytes were read into a buffer and written to a separate file. These files were then opened as the appropriate media type to ensure they had been read correctly. *OS_close()* was then tested on the open file descriptors to ensure they were closed properly.

Finally to test *OS_readDir()*, multiple directories were read, and then the resulting array of *dirEnt*'s were looped through and their *dir_name*'s were printed to ensure they had been read properly.

5. Conclusion

Navigating the FAT file system required careful adherence to the file specification documentation. Since all the information is packed so tightly and precisely, small errors in calculations could result in drastic errors, so it was crucial learning to debug with the hex editor to view values and using C properly to manipulate at the byte-level. Ultimately, a more thorough understanding of the FAT file system and its organization was achieved through this assignment, as well as following cluster chains and read operations.