Write up
Comments
Code

Operating Systems Homework #4

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to create a write 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 write and delete operations. 5 functions were implemented:

- *OS_mkdir(const char *path)* – makes a new directory at the specified path
- *OS_rmdir(const char *path)* – removes the directory at the specified path
- *OS_rm(const char *path)* – removes the file at the specified path
- *OS_creat(const char *path)* – creates a file at the specified path
- *OS_write(int fildes, const void *buf, int nbyte, int offset)* – given a file descriptor to a previously opened file, writes nbytes into the file specified at a given offset.

2. Approach:

The implementation of this assignment was heavily based on the previous work done on Homework #3. Built on top of the code submitted for the previous assignment, a lot of essential structs and helper functions for navigating the FAT file system were already implemented. Bootstrapping off of these functions used to extract information from the boot sector and traversing through file directories, a minimal amount of additional code logic was necessary to implement the write functions required by this project.

However, some additional helper functions were written to aid in specific write-oriented features. Among these were:

- *findFreeCluster()* – scans the FAT table to find the first free cluster available for allocation
- *clearClusterChain()* – given the first cluster number of a file data volume, clears the data regions allocated along the cluster chain and clears the FAT table values
- *isDirEmpty()* – determines if a given a directory is empty (useful for the *OS_rmdir()* function)
- *writeDirEnt()* – writes a *dirEnt* struct to disk
- *setFatValue()* – sets a FAT table entry to a specified value

These helper functions were used throughout the implementation of the 5 API functions. Each of these API functions follows a similar pattern. First it is necessary to traverse through the file

system to reach the parent directory of the file or directory that is to be modified or created. Then, depending on what the API function is, a matching directory entry name is searched for and appropriate actions are performed. For example in *OS_mkdir()* and *OS_creat()*, a directory entry with a matching name of the file/directory to create is searched for. If one is found, then errors are returned, but if one is not found, an appropriate entry is added in the first available space. Conversely for *OS_rmdir()* and *OS_rm()*, if a directory with a matching name is found, then that directory entry is cleared, as well as its associated data regions, and errors are returned if the entry cannot be found. These similar functionalities result in similar code structures for each API function's implementation

OS_write() was more complicated to deal with particularly because it required simultaneous writing data to the disk file as well as allocating new clusters in the cluster chain and the FAT table. The structure of this code was based off of the *OS_read()* code implemented from last assignment since they both follow similar patterns of requiring action when reading/writing to the end of a cluster block. The difference in *write()* however was in searching for a free cluster block, allocating space for it in the FAT table, and then proceeding to write in the newly allocated space until all appropriate bytes were written.

An interesting discussion point includes how free cluster blocks are discovered. The documentation specifies that free cluster blocks are denoted with a 0x0000 present in the FAT entry for the corresponding cluster block. Thus the easiest way to find an available cluster block was to simply linearly scan through the FAT table and find the first cluster block that was free. Although this is not incredibly efficient, it was the simplest to implement for the purposes of this assignment.

Another interesting discussion point was the handling of free directory entry spots for each directory. Once directories are removed, 'holes' are introduced in the list of directory entries, and this was solved by marking the first character of the directory name with 0xE5 as specified by the documentation to indicate a free dirEntry space.

3. Problems Encountered

One major issue encountered was with *OS_write()* which was incredibly tricky to implement since it had to simultaneously write to disk while also allocating free clusters when necessary. The initial implementation of *setFatValue()* relied on the helper function *getFatValue()* to check each entry in the FAT table for a free cluster marker. However, the *getFatValue()* function opens a new file descriptor upon each invocation and scans through the FAT table from the beginning until the appropriate FAT entry is found. This resulted in O(n^2) runtime for *setFatValue()* which was horrendously slow, and this was remediated by performing the linear scan directly in *setFatValue()*. The *OS_write()* function was completed only after careful debugging through printing allocated cluster numbers.

Another problem encountered was forgetting to close real file descriptors to the raw disk file. At some points, too many file descriptors were being opened, causing undefined behavior to occur and parts of the disk resetting to strange values. This caused strange behavior in the entries for the FAT table, where some entries that were previously marked as allocated suddenly became marked as free, causing the *write()* function to fail. This was fixed after careful debugging and closing file descriptors when appropriate.

Finally another problem encountered was dealing with the 'holes' in the directory entry lists after remove API functions had been called. In the original implementation for *OS_creat()*

and *OS_mkdir()*, the first available space was utilized for the new dirEntry, but it was realized that a duplicate entry name that came after the free space would not be detected (since this condition should cause the create function to fail). This was remediated by first looping through the whole directory and searching for duplicate names before creating the actual entry.

4. Testing

To test these library functions, a test program was created that linked the shared library, and each function was extensively tested. All API functions with the exception of *OS_write()* require traversing down the file path and have errors associated with files of that name being found/not being found. These different cases were all tested thoroughly such that all error and success conditions were triggered, and the appropriate response was noted. Additionally, the resulting raw file was examined at different stages using a hex editor to ensure that appropriate values were being written in the correct cluster locations and places within the FAT file system.

To perform an integration test, the write API functions were called together with the successful read API functions implemented in the last assignment. Folders and files were created and written to, and then the read API functions attempted to read back the correct information. This was how *OS_write()* was tested by first reading an image file and then writing this data to a newly created file. The data was then read back from the newly created file and saved to verify that the image had been properly copied, written, and read.

5. Conclusion

Creating these write API functions was significantly easier through bootstrapping from the previous code in the previous assignment. The helper functions that aided in navigation around the file system were tremendously useful in being able to start coding functionality sooner for the write API functions. Additionally, having prior familiarity with the system and the documentation was a huge help in understanding what needed to be done for each function. Overall, this was a useful assignment in extending prior knowledge of the FAT file system and carefully manipulating byte-level data without error.