Wright Kim and Neeraj Gandhi
Project 4 | File System Implementation
Project Report

Overview
The goal of this project was to create a file system using a single physical file on the Debian OS that we use for the class. This involved the implementation of a number of different functions, as detailed in Table 1.

Table 1: Description of Functions Needed

Function Name	Function Description
make_fs	Creates new file system on virtual disk
mount_fs	Mounts files system on virtual disk, makes files in it ready to use
dismount_fs	Dismounts file system from virtual disk, writes all data back so that disk shows any changes made
fs_create	Create new file
fs_open	Open file for reading/writing
fs_close	Close file, disabling read/write access
fs_delete	Deletes file from directory, frees data blocks
fs_read	Reads specified number of bytes from open file
fs_write	Writes specified number of bytes from a buffer to file data
fs_get_filesize	Returns length of file
fs_lseek	Changes file pointer relative to current location
fs_truncate	Shortens length of file, freeing unneeded data blocks in process

Implementation Strategy
The strategy we employed in trying to address the many facets of the project was to break down the problem. We examined each function to see what sequence of implementation would make the most sense. For instance, we realized that make_fs and mount_fs needed to be written and tested prior to writing or testing any of the other functions.

Design
We designed several different structs to contain the data that we needed to implement the file system.

The OFT was structured as follows: struct oftEntry = {char status; int offset, int index}. The status bit indicated whether that location in the OFT was available if someone tried to open another file. Only four files could be opened at a time, so this became an important design constraint. The offset value indicated how many bytes into the file the file pointer was. Lastly, the index provided the directory entry where other details of the file could be obtained.

The directory struct contained: {char status, char firstBlock[2], char fileName[4], char fileLen[3], char unused[6]}. The status bit here indicated whether a file existed or not. A directory can only have eight files in this file system, so if all eight entries in the directory have a status bit of value '1', a new file cannot be created. The firstBlock array contained the block number in memory where the data at the beginning of the file could be found. The fileName array consisted of four characters becausee files were not allowed to have names longer than four characters in this architecture. File length could be as much as 512 bytes, so three characters were provided to the fileLen array in the struct. The directory was linked to the FAT using the firstBlock value.

The FAT contained 32 entries in it, the index of which represented block number. Each entry consisted of a FATEntry struct: {char status, char blockNum[3]}. The status bit indicated whether the block of memory was in use by a file. Blocks of memory were not allowed to be split between files. The blockNum value indicated where the next block of the file was. That is, Block 1 could point to either Block 2 or Block 31, so files did not have to be stored contiguously in memory. If the blockNum value was “eof”, the end of the file had been reached.

Finally, the FAT was linked to physical block numbers. The FAT index was one-to-one mapped to data blocks; Entry 1 in FAT was linked to Data Block 1, Entry 2 to Data Block 2, etc. Each data block consisted of 16 characters in an array.

The mount_fs operation loaded the entire file system from disk. That is, all 1024 characters were read into arrays of the above-described structs. No operations except for mounting and dismounting dealt with disk operations.

Generally speaking, files were created and opened using a first-fit strategy. For instance, if the status bits in the directory were in the sequence {1,0,0,1,1,1,0,0,0}, then a new file would occupy index 1, meaning the status bits would look like {1,1,0,1,1,1,0,0,0}. Similarly, the OFT would place file descriptors for open files into the first open slot it could find. If open slots could not be found, then the file was not opened. This provided limits for what the file system could do simultaneously.

Another feature was the use of a super block. This block contained addresses at which the directory, FAT, and data could be found. However, because we were working using a single architecture, we ended up hard-coding the block numbers in many cases, instead of reading block numbers from the super block.

Memory Breakdown

The virtual disk that we were working with could contain 1024 bytes, and in our case this translated into 1024 characters. The memory was divided into blocks of 16 bytes each, and in some cases 16-byte blocks were subdivided, as with the FAT and directory. There was a single super block containing the address bounds of the directory, FAT, and data (this is Block 0, the blocks were 0-indexed). The directory was contained in the blocks 1-8, the FAT was in blocks 9-16, and the data was in blocks 32-63. This meant that blocks 17-31 went unused, but for the purposes of this particular file system that did not end up mattering. For a fully-functional system, we might be able to come up with some strategies to use more of the space that is available.