Christopher Richard Bilger
CIS 3207 - Professor Kwatny

Lab 4a

1. Virtual Disk File Partition

a.

BOOT SECTOR	FAT	DIRECTORY LIST	ROOT DIRECTORY	DATA BLOCKS (FILES AND DIRECTORIES) - 512 Byte Blocks

b. The size of the virtual disk file will be 10 MB. Since the size of the disk file must be at least 2 MB, a size of 10 MB should be perfectly sufficient for files up to 65 KB.

c. The size of the blocks are each 512 bytes, therefore there will be 10 MB / 512 bytes = 19,531 complete blocks.

d. Virtual Disk File Struct:
   i. struct VDisk {
   ii. struct Header *header; // header information about the virtual disk file
   iii. // The above might not be needed since it is contained in the BootSector
   iv. struct FAT *fat; // pointer to the beginning of the FAT
   v. struct RootDir *rootdir; // pointer to the root directory, which is made when the VDisk is first initialized
   vi. struct DirectoryList *dir_list; // pointer to the directory list
   vii. }

2. General Design

a. I will be using FAT as the structure of the virtual disk file. The main difference between FAT and inode is that FAT uses tables stored in a single, centralized location; whereas inode uses tables that are spread throughout the virtual disk file, and can change location when a file is created and/or deleted.

b. When a new file is created for a file is added that exceeds its allocated block size, the virtual disk file will need to accommodate with new blocks. This will be done with the file-allocation table. This centralized table will store the specific blocks that all files are located in and can edit. This will prevent files from changing blocks that they do not have permission to change.

c. There are two cases for storing files in this simple FAT virtual disk file. The first case is, of course, when a file is only one block (or smaller) in size. The file will simple be stored in that block and nothing else needs to be done. The second case, when a file is any larger than one block, is a bit trickier. The file will be linearly cut up into chunks of 512 bytes until the end of the file is reached. When we are assembling the chunks of the file, we will linearly go through all of the blocks that the file resides in and simply stitch them back together into a temporary file (in memory), which we will then be able to either edit or use for some other purpose.

d. In order to distinguish normal files (txt, pdf, etc.) from directory files, a flag bit will be used. For instance, if the flag bit is not set (0) then the file is a directory; however, if the flag bit is set (1) then the file is a normal file.

e. Keeping track of free, unused blocks can be done in a memory-saved list. Specifically, I think that my implementation of this list will save all used blocks as well as the total number of blocks. Then, whenever a file needs another free block to use, the program will look through the total number of blocks and check to see if they are in the "used list." If the block is in the "used list" then skip, otherwise change that block to used (by adding it to the used list) and use it for the file that needs it.

f. The root directory will reside in the lowest memory blocks. These memory blocks will contain the required contents in order to initialize and setup the virtual disk file.

3. Data Structures

a. Struct of FAT list:
   i. struct FAT {
   ii. unsigned int total_blocks; // Example of 20,000
   iii. unsigned int used_blocks; // After initialization might be 500
   iv. HashMap table[total_blocks]; // The essence of the FAT list
   v. // The above is a bunch of key-value pairs
   vi. }

b. Struct of directory entry:
   i. struct Entry {
   ii. char filename[128]; // Filename, excluding file extension
   iii. char file_extension[32]; // The extension type of the file

iv. char directory; // byte flag for file/directory specification
v. unsigned int starting_block; // the location in the virtual disk file to start
vi. unsigned int size; // size in bytes of the entire file
vii. unsigned int offset; // Current offset in the file
viii. // Further attributes can be added at a later time, such as time and date stamps, etc.
ix. }

c. Struct of a single block:
    i. struct Block {
    ii. unsigned int next_block; // The FAT table entry of the next block, if any
    iii. char data[512]; // The actual data to store in the block
    iv. }

d. Struct of the directory list:
    i. struct DirectoryList {
    ii. unsigned int dir_count; // The total number of current directories
    iii. struct Entry *all_dirs; // A list containing all directory files
    iv. }

e. Struct of the boot sector:
    i. struct BootSector {
    ii. unsigned int bytes_per_block; // Will be 512
    iii. unsigned int reserved_blocks; // Will be the first X blocks used to initialize the virtual disk file
    iv. unsigned int total_blocks; // Will be calculated. Ex: 10 MB / 512 bytes
    v. unsigned int filename_size; // 128
    vi. unsigned int file_extension_size; // 32
    vii. char disk_name[128]; // The name of the virtual disk file
    viii. }

4. Pseudocode

a. The main logic will center around a shell-like loop that will ask for user input on what the VDisk should do next, implementing those user inputs into the VDisk, and then looping back around for user input.
    i. Each of the structs will need functions that will be able to manipulate them. The function calls, argument types, and return types will be in the associated file aptly named “struct_functions.pdf”. There will also be

".odt" and ".docx" versions available in the same location, with the same filename of "struct_functions".

b. The creation, allocation, and destruction of all data structures will also we available in the above file.

5. Testing

a. Some possible bugs that I think will occur, as is usual in C, are memory related bugs. With the use of many new structs and pointers to all of these new structs and lists, I can foresee some memory related bugs that I will need to think about while programming the virtual disk file.

b. Some modular tests that I will implement are tests on the following:
   i. Creation of a VDisk
   ii. Initialization of the VDisk
   iii. Creation of a new file
   iv. Editing a file
   v. Deleting a file
   vi. Creation of a new directory
   vii. Editing files in the new directory
   viii. Deleting files in the new directory
   ix. Deleting a directory itself (making sure that all sub-files and sub-directories are deleted as well)