Machine Problem 7: Vanilla File System
CSCE 611, Hung-Hsueh Shih, UIN 132005330
Object:
Implement a simple file system that has files that support sequential access only.
I have modified the following files.
file_system.H
file_system.C
file.H
file.C
In file_system.H, I have two classes 1. Inode 2. FileSystem
In Inode class, I have added several variables to store the information for the file.
I added (1) a pointer point to the file address, (2) an integer indicates the block is used to store this file, (3) an integer indicates the current position of the file, (4) an integer indicates the size of the file, (5) a pointer points to the file system.
Besides, I have added two functions (1) inode_read and (2) inode_write in order to write the inode list into a disk.
In the file system, I have added two variables, (1) number_of_free_blocks and (2) number_of_inodes, the first one indicates how many free blocks are left in this file system, the last one indicates how many inodes are stored in the nodes list.
In addition, I added several functions, (1) free_block_read() , (2) free_block_write, (3) GetFreeInode(), (4) GetFreeBlock() I'll illustrate these functions in the following implementation part.
Implementation for a file system call, I modified the following functions in file_system. C
inode_read(): I read the first block into the buffer.
Indoe_write() Write the first block into the buffer.
FileSystem() constructor: I initialize the inode list with a maximum number of an inode in one block, then set the variable in the file system to 0.
FileSystem() destructor: I store the inode list and free block list back into the disk
Free_block_read(): read free block list from disk
Free_block_write(): write free block list back into disk
Format(): I initialize the disk pointer and size for the file system. Besides, I clean up all data in this file system then initialize the free block list, which marks the first block and the second block are used then set the rest of the free block list to free.
Mount(): first of all, I check whether the filesystem has been initialized by using a flag ->count, if the count is 0 means the file system has not been set up. Therefore, I initialize the file system by connecting it to the disk and writing both lists(inode list and free block list) into the disk.
GetFreeInode(): return the next number of a free inode in the inode list.
GetFreeBlock(): return the next index of free block in the free block list.
LookupFile(): iterating the whole inode list and checking the id of each inode object which represents the file name to see whether the file exists or not.
createfile(): fist of all, check whether the file with the given name has been created, if so, return an error, else I receive the next free block in the disk and the next free inode list by calling GetFreeBlock() and GetFreeInode(). After that, I initialize the inode object by assigning the id, current_ block, file_size, and position. In the end, I update the inode list and free block list and write these two lists into the disk.
DeleteFile(): First of all, I check whether the file exists, if not, return false, else, I iterate the inode list to find out the file by checking the id. After I find the file, I delete the file object by deleting the file pointer I store in the inode then I delete the inode object and move all items in the inode list one poison forward. For the free block list, I find out corresponding block is used to store this file and mark this as free. In the end, I write both lists back into disk.
For the File class, I have modified the following in the file. H

I add several variables in file class, the pointer point to the corresponding inode object, the pointer point to the corresponding file system, the integer represents the current position in the file, the integer represents the block is used to store this file, the name of this file, and the size of this file.
Implement for file class, I have modified the following functions in the file.C
File() constructor, I initialize the variables for file, point the file system, inode, id, and the, in the end, I find the corresponding inode object in inode list and update the value and point this file to file address which is stored in inode object.
File() destructor, I update the file information into inode and write inode list and free block list into a disk.
Read(): I read the corresponding block which is used to store files into the buffer. Then copy char one by one store into the buffer. At the same time, I check whether the current_position is reach the file size or reaches the end of the file. Return how many chars I have read from the buffer.
Write() I read the corresponding block which is used to store the file into the buffer. Then I copy the char one by one into the buffer. I check whether I should extend the size of the file or there is no space to write in. In the end, I write the buffer back into the disk.
Reset() : I reset the current position in to 0
Eof(): I check whether the current position is the end of the file.
Bonus:
OPTION 1: DESIGN of an extension to the basic file system to allow for files that are up to 64kB long.
For the implementation mentioned above, the maximum size of a file is 512 bytes, that is, one file can only store in one block. In order to support files up to 64 KB long, we have to let files get more blocks. The easiest way is we check we store the information in inode about how many blocks we use for this file and the current position at which block. For the file itself, in order to keep the sequence of the files to the same, we can simply use a linked list in a file, for every last four char in the file we use these four bytes to store the next location for the file. That is to say, every time we read or write a file, we need to check not only we reach the end of the file or there's still have remain files stored in a different block.
For the Inode class, we need to add the following variables.

1. total_blocks_for_this_file
2. current_block_for_this_position
3. first_block_for_this_position

For the FileSystem class we need to modify the following functions
Delete file(): since we support more than one block more each file, we need to iterate the whole inode list and find the file which has the same id and release the memory, besides for the free block list, I need to find out the corresponding blocks and release the memory as well.
For File class we need to add following variables,

1. total_blocks_for_this_file
2. current_block_for_this_postion
3. first_block_for_this_postion

For the File class we need to modify the following functions
Read(): for the read file read function, since we need to know the address of the next file, we can simply use the last four char to store the address of the next file, when we read a file, we start at the current position, and see whether we reach the end of file, reach the end of this block, or reach the maximum size of the files, that is, 64KB. To be more specific, if we reach the end of the file, we do not need to read anymore, can simply return the number of char we read, if we reach the end of the block, we should check whether an address at the end of file, if it is, we need to read the next block from disk and keep going. In the end, if we reach the maximum size of the file, we need to stop.
Write(): since we support files ups to 64KB, we need to check whether we write all data we want or, whether this block is full but we still want to store more, if it's the last situation, we need to call the find_next_free_block function in the file system to find out next free block, at the end we can keep storing data into the disk.