Implementation of Simple File System

Vishruth Raghuraj Gollahalli

(Bonus attempted: Option 1: Design of extending file system to handle longer file sizes rather than one block file size)

Part 1: File

File operations implemented were read, write, end of file check and reset.

1. Read: To read a file, first a buffer was initialized. In a for loop, the characters from the disk were read one by one from disk using the underlying SimpleDisk read operation by specifying the block number and the buffer. For each character read, the character was copied from the initialized buffer to the read buffer (_buf). After read was complete, the number of characters read, _n, was returned.

2. Write: The write operation was done similarly to the read operation. Main difference was that in write, the initialized buffer data was copied to the disk buffer and then written to the disk using the underlying SimpleDisk write operation.

3. Reset: The reset function simply reset the position to the beginning of the file by marking the file_pos to 0.

4. End of file: End of file was checked by simply checking if the file_pos was greater than the block size. This works because the maximum file size is equal to the size of one block.

Part 2: File system:

Inode list was implemented as a linked list with an id filed and a next pointer.

To create a file, first the inode list was checked whether there existed a node with the same id as the file requested to be created. If there was, then an error message was raised, else a new inode node was added to the list with the new file id.

To delete a file, the linked list of inodes was traversed. If no file id of the requested file to be deleted was found, then an error message was raised. Else, this node was removed from the inode list.

Lookup for a file was carried out by traversing the list of inodes.

I was unfortunately unable to implement it fully and correctly because I was unable to implement the setting and unsetting of the free_blocks list. The initial plan was to mark a char in the free_block with the index corresponding to the data block number as being

marked with a 0xFF if used and 0x00 if unused. I tried with this approach but was not able to fully implement the file system. I was also unable to find a way to store inodes in block[0] and free list in block[1] which was my design plan for implementing the file system.

Bonus attempt: Design of larger file sizes:

For file operations:
In the case of file operations the following modifications need to be made for the read and write. If the number of characters we need to read are greater than the block size, we need to read the first block fully and increment the position within the file. If this position reached the end of a block, we retrieve the block that holds the remainder of the file. To keep track of this, we can use another data structure that would store all the block indices for one file. Once one block is read, and the file is still not read completely, we retrieve the next block and continue until the end of file is reached. Then, all these blocks would need to be marked as not free and the number of free blocks would be decremented.
Similarly for the write operation, we need to write until the block is full. Once it is full and we have more to write to a file, we create a new block entry, store this block in the new data structure for this file and continue writing to the new block.

For file system:

The inode list would need to store, along with the file id and the next pointer a list of all blocks where this file is stored. This list can be used as the data structure described earlier in file operations to carry out file reads and writes.

We would need further functions such as getMoreBlock() that would retrieve a new block to write file contents into in case the file size was greater than one block. This new block would be kept track of in the inode of that particular file.