Brandon Walton, Bibhushita Baral
Dr. Aisha Gombe
CSC-4103, Operating Systems
3 December 2023

File System Design

Structures:

- Our code is composed of two structures: **Inode**, and **FileInternals**

Inode
uint16_t **single_indirect**
FileMode **mode**
uint16_t **size**
uint32_t **position**
uint8_t directblocks[13]

Size of Inode: 28

FileInternals
char ***fsname**
short **inode_id**
bool **open**

Size of File: 8

- The Inode is used to store metadata such as the block to where the data is pointed to, the file's size, the file's current position, etc. The FileInternals stores information referencing the file such as its name, state, and inode_id.

Global Fields:

uint16_tinode_bitmap[SOFTWARE_DISK_BLOCK_SIZE]	Bitmap that tracks available inodes
uint16_tdata_bitmap[SOFTWARE_DISK_BLOCK_SIZE]	Bitmap that tracks available data blocks
Inode *inode_array[SOFTWARE_DISK_BLOCK_SIZE]	An array that stores all Inodes
File directory_array[SOFTWARE_DISK_BLOCK_SIZE]	An array that stores all File_Internals

Notes on inode_array and directory_array:

- These two arrays store all the inodes, both existing and non-nonexisting, and all File_Internals.

- The idea is when a function requires an inode, the inode_array can be read from its corresponding inode block and copy all the contents back into the inode_array
    - Ex:

```
directory_array[directory_index]->open = false;
    inode_array[inode_index]->mode = mode;
    write_sd_block(inode_array, i_block);
```

    - The inode_array[inode_index] refers to the inode that is currently being used, and when finished with the operation, the whole inode_array is written back to its corresponding inode block to ensure that the modified inode, [inode_index], is stored. The same is done for directory_array.

Determining Blocks:

- The blocks were determined by the provided outline given by Dr. Gombe, and to ensure that the blocks were correct, we indexed the blocks as follows.

- **Inode_Block:** Since inode_id's span from 0-127 and inode block span from 1-5, the integer from the formula below shows which block it is allocated to. Note: INODES_PER_BLOCK = 128.

```
return (inode_id / INODES_PER_BLOCK) + FIRST_INODE_BLOCK;
```

- **Directory_Block:** The directory block was determined by blooping through the directory array and searching for File's name. The while loop started from the first directory block and lasted until the last directory block. Within the while loop, a for loop existed that went from 0 to the maximum number of Directory entries. Once the desired element was found, the block number was returned. A similar approach was also done for retrieving the directory_index, and inode_index.

```
short get_directory_block(char *name)
{
    short potential_block = FIRST_DIR_ENTRY_BLOCK;
    while (potential_block <= LAST_DIR_ENTRY_BLOCK)
    {
        read_sd_block(directory_array, potential_block);
        for (short i = 0; i < DIR_ENTRIES_PER_BLOCK * (LAST_DIR_ENTRY_BLOCK - FIRST_DIR_ENTRY_BLOCK); i++)
        {
            if (directory_array[i] != NULL && strcmp(directory_array[i]->fsname, name) == 0)
            {
                write_sd_block(directory_array, potential_block);
                return potential_block;
            }
        }
        potential_block++;
    }
    fserror = FS_OUT_OF_SPACE; // Couldn't find block
    return -1;
}
```

- **Data_Block:** The data block was determined by using the current file's position and dividing it by the size of the block. Then add the outcome to FIRST_DATA_BLOCK to offset the elements. With the result, the integer was the block of data that the current position could read or write from.

```
short data_block = (inode_array[inode_index]->position
/SOFTWARE_DISK_BLOCK_SIZE) + FIRST_DATA_BLOCK; // grabbing current block
based on position
```

Direct Block Indexing:

- To ensure that our system works with the provided block arrangement, the size of our structures should add up to be 32 bytes. With this consideration, our direct block array had to be of size uint8_t to ensure the alignment of our structures. With this, our direct block could only hold numbers 0-255, however, the data blocks were from 70-4095.
- **Problem:** How can we have an array of uint8_t that can map to the numbers 70 - 4095?
- **Solution:** To determine the numbers that can be stored for direct block entries, first the range of blocks we could store was 70 + sizeof(uint8_t) = 70 + 255 = 325. So blocks, 70-325 represented the direct blocks. Then we began indexing as follows:
-

70 - 325	70	71	72	73	74	75	76	77	78	79	…	324	325
0 - 255	1	2	3	4	5	6	7	8	9	10	…	255	0

- 70 was mapped to 1 to allow the loop to not get 0 confused with empty, and 325 was manually mapped to ensure that 0 would not get confused with empty.
- For example, whenever data was being written from block 75, the directblock array would store the value of 6, and later when the data was read, the 6 would be mapped back to the 75, so the read can be performed from block 75.

Indirect Indexing:

- In case a file has used all of its direct blocks, a single indirect block was used to store additional files. The data was stored as follows:
    - **Step one:** The block at which the write position was used for the single indirect block.
    - **Step two:** A dynamic array of uint16_t was then read from the block to grab the most recent information, if it was the first read the memory was manually set to 0. In this array, the index was determined by finding a free data block from the data_block bitmap.
    - **Step three:** We continued to write as normal, and when the write was finished the index at which the free block was found was then written back into its corresponding block. After the index was written, the whole single

_indirect_array was stored back into the indirect block's single_indirect_array was then stored back into the single indirect block's block.
    - **Visual Example:** User writes "Hello" in block 83. However, all direct blocks are full, and the single indirect write happens at an arbitrary index i which holds the next free block, 84.
    - Flow: Single_Indirect{83} -> single_indirect_array[i] -> data_block{84} -> "Hello"

System Limitations:

- Due to our implementation, some limitations have been established. They are found below
    - **Separate file referencing:** due to the nature of storing and tracking inodes and directories, whenever a file is created in a test case and is referenced in another test case, unexpected errors occur. Such as not updating or displaying information, consistently.
    - **Return Size Issues:** Since our size is a uint16_t and our position is a uint_32, issues with mapping the elements to return the size have unexpected behavior after size of (uint16_t). However, the functionality of the program continues to work.