CSCE-611: Operating Systems Machine Problem 7

Name: Vamsi Tallam UIN: 432001932

Files modified/added:

1. file.C
2. file.H
3. file_system.C
4. file_system.H

Design Steps:

The following are the changes made:

1. **FileSystem.C/H:**
    a. **File System Constructor:** Allocate memory (on the heap) for the inode list and the free blocks list.
    b. **FileSystem Destructor**: Destructor method for FileSystem class has been implemented to clear the inode list and free blocks list
    c. **Mount**: function loads inodes, free block list into memory from the disk
    d. **Format**: Clear the free blocks list to clean all the memory and to initialize the inode list to mark files that doesn't exist
    e. **LookUpFile**: Used to find the node of the file from the inode list by matching the file id. So, an address is returned. Inode* is returned.
    f. **CreateFile**: Identify a free block and then assign it to this file. Then update the corresponding inode in the inode list with all the attributes. Fill all the details, also the inode list and updated free blocks to the disk to reflect the most updated information for successive file creations
    g. **DeleteFile**: finds the relevant block and adds it to the free blocks. Also, update the inode list to reflect the latest file status. Now that we have deleted, we need to push the inode list and free blocks to the disk again to allow for successive file creations and deletions

2. **File.C/H:**
    a. **File Constructor:** Used to initialse the attributes. Loops through the inode list to get the relevant information. Cache gets the file by reading into the file
    b. **File Destructor:** Write the cached data from the previous write to the disk before cleaning up to disk and deallocate all the memory

c. **Read:** Cache all the required number of characters by reading from the disk and then update the current position till the last read operation
d. **Write:** Write all the cached characters to the disk and update the current position of the file
e. **Reset:** Update the file position to 0
f. **EOF:** Used to check if the current position is equal to the block size where in we need to change to the next block or potentially the end of the whole block.

Bonus: OPTION1

1. The current file system supports only 512 bytes of data which is a single block. So, to support files upto 64kB, we need at least 128 blocks. The concept of linked block allocation can be explored to extend the memory.
2. Hence, along with the block number and current position maintained in the basic implementation, we need to keep track of metadata to maintain the consecutive blocks.
3. The following structure helps in giving a design idea on how the implementation can be taken advantage of

```
Class LargeFile{

        int startingblockNum;     // the current block number
        Block* ptr;               // ptr to next block
        int totalNumOfBlocks;   // Total num of blocks this file requires
        int currentPos ;          // Keeps track of position to easily navigate to the
        next block useful while reading.

};

Block block{

    int diskPtrPos;         // contains the position

    int nextBlockOffset; // contains the offset by which the next block can be located

    bool EOF = true/false;  // depending on whether it is the last block or not

    Block *nextBlockptr;    // contains a ptr to the next block in series

};
```

4. Now that we have a basic thing that keeps track of 64kB long file sizes, we need a mechanism to properly read and write the data. The data in 512 bytes is always cached into the cache block defined, we need to individually read each of the 512 bytes and then simultaneously give out the data or write the data to the disk during the write operation to the disk.

5. Now to properly read the linked blocks into the cache block, we need a mechanism to easily get the linked blocks, hence we can make use of the current position stored in the Block to offset the pointer once the end of the current block is reached. Also, the **Inode** contains the file size, hence by easily getting the information related to the number of blocks in the file we can actually loop through that many number of blocks and by using the current position, we can read or write each 512 byte block into cache and hence we can give out or write to the disk respectively.
6. Now, we need a mechanism to identify which of the blocks have been utilized, so the concept of contiguous block allocation used in MP2 can be extended to manage the files.
7. Once the file is deleted, we can clearly use the metadata in the Block to clear the used blocks and can release the memory and add to the list of free blocks.
8. The actual implementation was completed, and the results are attached below. I also tried to implement **option 2** of bonus, but due to other projects and exams, I couldn't complete it.

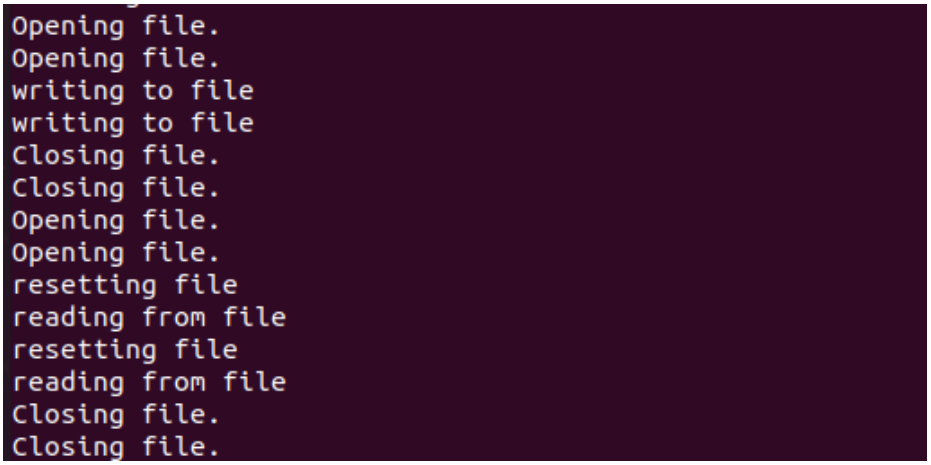
Results:

1. The basic initialization data like formatting, mounting

Creating a new file, opening, writing and closing after writing

```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
In file system constructor.
Hello World!
formatting disk
Done formatting disk
mounting file system from disk
creating file with id:1
creating file with id:2
Opening file.
Opening file.
writing to file
writing to file
Closing file.
Closing file.
```

2. Opening a file, writing resetting, reading and closing



3. Deleting a file. As seen in the image, to ensure that the file has been deleted, lookupfile has been called again after deleting to confirm the deletion and we can clearly see that look up file is unable to find the file after deletion which is expected.