MP7: Simple File System

Sushant Vijay Shelar
UIN: 733001479
CSCE611: Operating System

Assigned Tasks

Main Part: Completed.

Introduction:

- The document outlines the architecture and development of a Simple File System, primarily centered on the creation of two key classes: Filesystem and File.
- It delves into the details of these classes, their functions, and interactions within the system.
- Additionally, the document addresses the first bonus option presented in the MP7 handout, which involves the conceptualization and design of an enhanced file system capable of supporting larger file sizes.
- This extension is discussed in terms of its feasibility, potential design modifications, and the impact on the overall system performance and storage management.

Code Description:

Files changed:
• file.C • file.H • file_system.C • file_system.H

File System Class

- The File System class plays a pivotal role in overseeing the operations of reading, writing, and structuring files on a disk.
- Its responsibilities encompass a range of tasks, including allocating disk space for new files, deleting existing files, and optimizing the disk's layout to ensure swift file access.
- Central to this class is the inode class, which the File System class utilizes to represent disk files.
- The inode class encapsulates critical file details, such as the file's unique ID, the disk block storing the file's data, and the file's size.
- Additionally, the File System class is equipped with functionalities to mount and format the disk, alongside capabilities to search and retrieve files stored on the disk.

```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    //assert(false);
}

FileSystem::~FileSystem() {
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */
    //assert(false);
}


/*--------------------------------------------------------------------------
/* FILE SYSTEM FUNCTIONS */
/*--------------------------------------------------------------------------


bool FileSystem::Mount(SimpleDisk *  disk) {
```

Figure 1: File System Class

Inode Class:

- This class contains various private attributes and is closely linked with two friend classes: FileSystem and File.
- The private members of the Inode class encompass essential elements like id, first block, file size, and its associated file system, along with other possible data.
- The class is also designed with additional functions that facilitate the reading and storing of inodes on the disk. Below is the outlined structure of the Inode class

```
class Inode
{
  friend class FileSystem;
  friend class File;

private:
  FileSystem *newFileSystem;
   long id;
   long file_size;

   long first_block;
};
```

Figure 2: Inode

Management of INODE and FREELIST:

- In this system, Block 0 is designated for storing inode information, while Block 1 is allocated for the storage of Freelist blocks information.
- The management of inodes and the Freelist of blocks involves specific strategies. For instance, the maximum number of inodes is defined by the static constant MAX_INODES, calculated as SimpleDisk::BLOCK_SIZE / sizeof(Inode).
- The inodes themselves are represented by an Inode pointer, *inodes. For the management of free blocks, the system employs an unsigned character array, *free_blocks, instead of a traditional bitmap approach.
- Each operation of reading and writing blocks is executed using a buffer with a size of 512 bytes

Comprehensive Implementation of the FileSystem:

- Mount(): This function initializes the fileSystem class variable with the disk.

```
bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts("mounting file system from disk\n");
    disk=_disk;
    /* Here you read the inode list and the free list into memory */

    return true;
}
```

Figure 3: Mount()

- LookupFile(): This function is designed to search the disk for a file identified by its specified Id. Upon finding a matching file, it returns the block number stored in the inode. If the file is not located, the function returns a null pointer.

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::putui(_file_id); Console::puts("\n");
    /* Here you go through the inode list to find the file. */
   // assert(false);

   unsigned char cache[sizeOfBlock];
   memset(cache,0,sizeOfBlock);

   disk->read(0, cache);

   inodes= (Inode *) cache;

for (int i = 0; i < MAX_INODES; i++) {
    if (_file_id == inodes[i].id) {
        Console::puts("File found \n");
        return &(inodes[i]);
    }
}
   return NULL;
}
```

Figure 4: Lookup File method

- Format(): It initializes a blank file system after clearing the disk.Block 0 (the inode list block) and Block 1 (the freelist block) are initialized here. The status of blocks 0 and 1 is then changed to USED (set to 0).

```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) {
    Console::puts("formatting disk\n");
    //assert(false);
    /* Here you populate the disk with an initialized (probably empty) inode list
       and a free list. Make sure that blocks used for the inodes and for the free list
       are marked as used, otherwise they may get overwritten. */

    unsigned char cache[sizeOfBlock] ;

    int num_blocks = _size/sizeOfBlock;

    memset(cache,0,sizeOfBlock);

    for(int i=0;i<num_blocks;i++)
    {
        _disk->write(i,cache);
    }


    memset(cache,0, sizeOfBlock);

    _disk->read(0,cache);

    Inode *inode = (Inode *) cache;

    for(int i=0;i< MAX_INODES; i++)
    {
        inode[i].id = -1;
        inode[i].first_block=-1;
        inode[i].file_size=0;
    }

    _disk->write(0, cache);


    memset(cache, 0, sizeOfBlock);

    _disk->read(1, cache);
```

Figure 5: Format method

- CreateFile(): This function is tasked with establishing a new file identified by a specified Id. Initially, it verifies the existence of the file Id within the inode list. In cases where the file already exists, the function returns 'false' as a response, signaling the pre-existence of the file. However, if the file Id is not found in the list, the function proceeds to allocate the file Id and a corresponding block number to the file in the inode.

```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts("creating file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */
    bool track = false;
    unsigned char cache[sizeOfBlock];
    memset(cache,0,sizeOfBlock);
    disk->read(0, cache);
    inodes = (Inode* ) cache;

for (int i = 0; i < MAX_INODES; i++) {
    // Check if operation is already completed
    if (track) {
        break;
    }

    // Check if the inode with the file ID already exists
    if (inodes[i].id == _file_id) {
        return false; // File already exists, no need to create a new one
    }

    // Check if the inode is free (i.e., its id is -1)
    if (inodes[i].id == -1) {
        // Assign file ID, allocate the first block and set file size
        inodes[i].id = _file_id;
        inodes[i].first_block = GetFreeBlock();
        inodes[i].file_size = sizeOfBlock;
        // Write the updated inode information to disk
        disk->write(0, cache);
        // Notify that a new file was created
        Console::puts("File created\n");
        // Set flag to indicate that the operation is complete
        track  = true;
        break; // Exit the loop as the operation is completed
    }
    // Continue to the next iteration if the current inode is not free
}
```

Figure 6: Create File Method

- Available(): Function is used to mark a specific block on the disk as free. It does this by first reading the freelist information from the disk into a cache, checking if the block is currently in use, and then setting it as free if it is. The updated freelist is then written back to the disk. The function returns true if it successfully frees the block, and false if the block was already free.

```
bool FileSystem::Available(int block)
{
    unsigned char cache[sizeOfBlock];
    memset(cache,0,sizeOfBlock);
        disk->read(1, cache);

        if (cache[block] == 0) {
            return false;
        }

        cache[block] = 0;
        disk->write(1, cache);
        return true;


}
```

Figure 7: Available Method

- GetFreeBlock(): Function searches for a free block on the disk, marks it as used, and returns its number. It reads the current block status from the disk into a cache, looks for the first block that is free (marked as 0), marks this block as used (1), and then updates this information on the disk. If it finds a free block, it returns its number; if there are no free blocks available, it returns -1.

```
int FileSystem::GetFreeBlock()
{
    unsigned char cache[sizeOfBlock];
    memset(cache,0,sizeOfBlock);
        disk->read(1, cache);

        for (int i = 0; i < sizeOfBlock; i++) {
            if (cache[i] == 0) {
                cache[i] = 1;
                disk->write(1, cache);
                return i;
            }


        disk->write(1, cache); // Perform the disk write after the loop if no zero found
        }
        return -1; // Return -1 if no zero found in the cache

}
```

Figure 8: Available Method

- DeleteFile(): The primary task of this function is to locate the file designated for deletion. Upon finding the file, it proceeds to reset the corresponding inode and changes the status of the file's block number to 'FREE' (indicating it as 0). This action renders the File Id and the block number irrelevant and no longer in use. The implementation of this function relies on the utilization of the FreeBlock() function to execute these changes.

```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("deleting file with id:"); Console::puti(_file_id); Console::puts("\n");
    unsigned char cache[sizeOfBlock];
    memset(cache,0,sizeOfBlock);
    disk->read(0, cache);
    bool track=false;
    inodes = (Inode *) cache;
    for(int i=0;i< MAX_INODES; i++)
    {
        if(track)
        {
            continue;
        }
        if(inodes[i].id != _file_id)
        {
            continue;
        }
        inodes[i].id = -1;
        Available(inodes[i].first_block);
        inodes[i].first_block=-1;
        inodes[i].file_size=0;

        disk->write(0,cache);
        //set track to true
        track = true;
    }
    return true;
}
```

Figure 9: Delete File Method

Testing:

I used the 'make' command to compile all the files in the "mp7" folder. Then, I ran a script called 'copy kernel' to copy the bin file to the disk. After that, I executed the code using the "bochs -f bochsrc.bxrc" command.

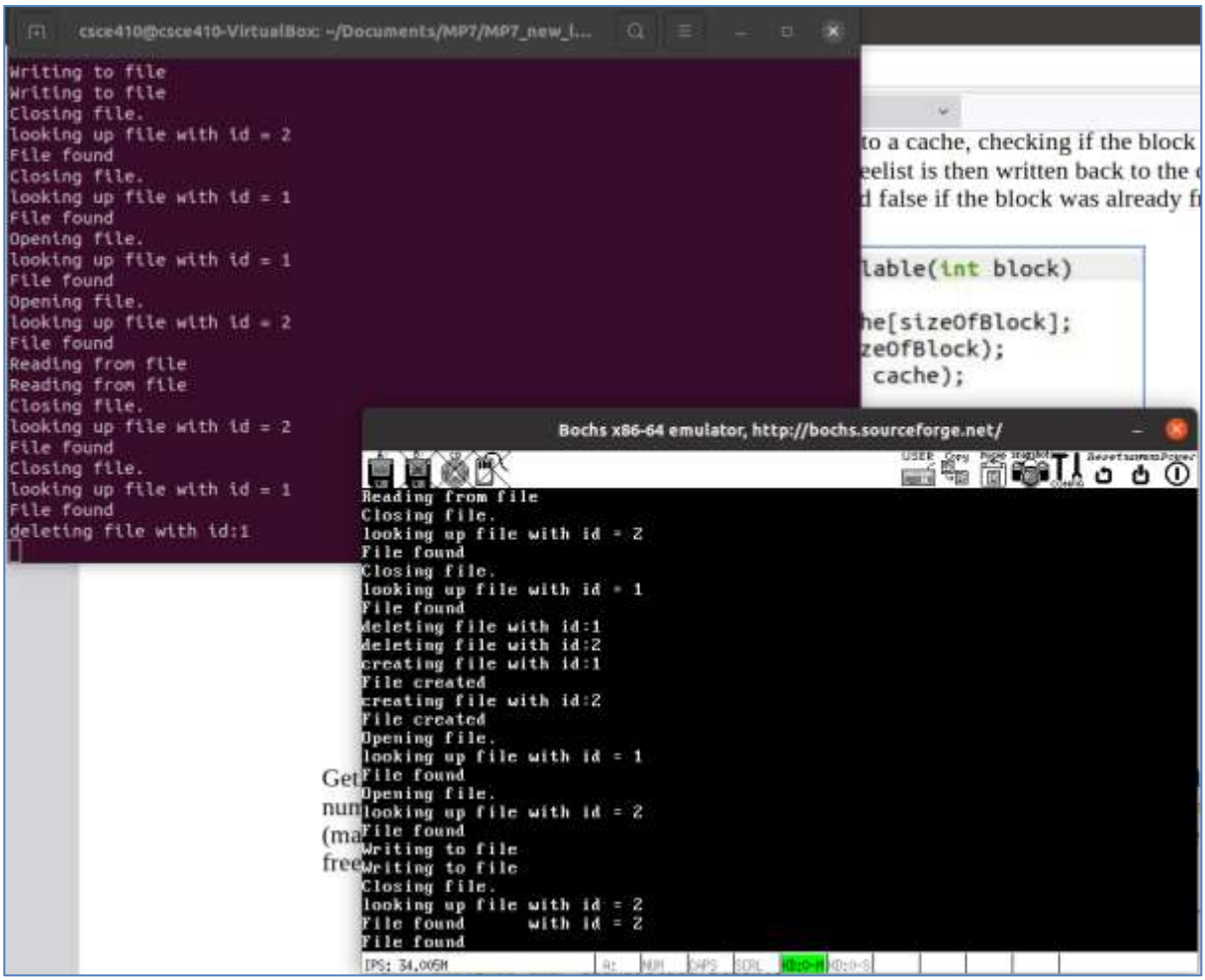

Figure 10: Testing