MP7: Vanilla File System

Nikhil Nunna
UIN: 525008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Option I: Omitted.
Option II: Omitted.

System Design

The goal of this program was to create a simple file system that could handle a couple of files along with read and write operations on those files. This was done by creating a couple of lists that contained information about the file system. Essentially all of the relevant information about the file system is in 2 lists the first of which is an Inode list, which contains information pertaining to a corresponding file. So, each file that gets created gets a corresponding Inode which carries meta data about it such as the length of the file, the memory block it occupies, and the file ID. The second list we have is a list of all the blocks of memory blocks we have it is called a free-block list but it contains the state of all the blocks of memory we have. This helps us find empty blocks to allocate to new files as needed and then set them to free as files get deleted. So in essences we handle all the creation, management, and delegation of files by manipulating these 2 lists. For actual read and write operations we find the block number the file is stored in using the Inode list and then we copy the block to a buffer, we do our reading and writing to the buffer and then right the buffer which basically cache back to the block.

Code Description

The code compiles and runs normally. All the grader needs to do is run 'make', './copykernel.sh' and then 'bochs -f bochsrc.bxrc'. As an alternative to this, I've created test.sh which can be run using './test.sh'. The test.sh shell script just runs make, copykernel and the bochs command for you.

test.sh: code block : This shell file compiles and runs the project

```
#!/bin/bash

make
./copykernel.sh
bochs -f bochsrc.bxrc
```

FileSystem.C: constructor : In this function, I initialize the inode and the free block list data structures.

```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    // allocate the inode list and free block list
    inodes = new Inode[MAX_INODES];
    free_blocks = new unsigned char[512];

    // mark blocks 0 and 1 as used
    free_blocks[0] = '1';
    free_blocks[1] = '1';
    // mark blocks as empty
    for (int i = 2; i < 512; i++)
    {
        free_blocks[i]='0';
    }

}
```

FileSystem.C: destructor : In this function, I save the Inode list and the free block list to the disk before I delete the objects.

```
FileSystem::~FileSystem() {
    Console::puts("unmounting file system\n");

    // Write the file system data to disk before destroying object
    disk->write(0,(unsigned char *)inodes);
    disk->write(1,free_blocks);
    // delete objects after writing them to disk
    delete[] free_blocks;
    delete[] inodes;
}
```

FileSystem.C: mount : In this function, I read the file system from disk so that my file system object can navigate the disk. This is done by reading the Inode and free blocks list blocks.

```
bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts("mounting file system from disk\n");

    /* Here you read the inode list and the free list into memory */

    //save the disk your mounting on
```

```
    disk = _disk;

    //inode block
    _disk->read(0,(unsigned char *)inodes);
    //free list block
    _disk->read(1,free_blocks);
    return true;

}
```

FileSystem.C: Format : In this function, I format this disk by adding my Inode list data structure and free blocks list data structure to it. I then destroy the temporary object I used to format the disk.

```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { // static!
    Console::puts("formatting disk\n");
    /* 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. */
    // Create Inode and Free blocks list for formatting.
    Inode * inodes_list;
    inodes_list = new Inode[MAX_INODES];
    unsigned char *  free_blocks_list;
    free_blocks_list = new unsigned char[512];
    // mark blocks 0 and 1 as used
    free_blocks_list[0] = '1';
    free_blocks_list[1] = '1';
    // mark other blocks as empty
    for (int i = 2; i < 512; i++)
    {
        free_blocks_list[i]='0';
    }
    // write inode and free block lists to disk
    _disk->write(0,(unsigned char *)inodes_list);
    _disk->write(1,free_blocks_list);
    // blow away these temp lists used for format
    delete[] inodes_list;
    delete[] free_blocks_list;
    return true;
}
```

FileSystem.C: LookupFile : In this function, I look thru the Inode list till I find one that is being used, I then check to see if it has the same file ID that was passed in when I find one with a matching file ID I return the address of the Inode.

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id="); Console::puti(_file_id);
    Console::puts("\n");
    /* Here you go through the inode list to find the file. */
    // loop thru inode list till you find one with the same file_id
    for(int i = 0; i<MAX_INODES; i++){
        // check if node is valid
        if (inodes[i].val)
        {
            // return the inode if it has the same id
            if (_file_id == inodes[i].id)
            {
```

```
                return &inodes[i];
            }
        }

    }
}
```

FileSystem.C: CreateFile : In this function, I create a file by first ensuring there isn't already a file with the same file ID. Then I find an unused inode which I then assign an unused block too along with the file ID. This essentially creates a place for the file and all of it's meta deta.

```
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. */

    // loop thru inode list to ensure there isn't one with the same ID
    for(int i = 0; i<MAX_INODES; i++){
        // check if node is valid
        if(inodes[i].val)
        {
            // return false if it has the same id
            if (_file_id == inodes[i].id)
            {
                return false;
            }
        }

    }


    // save position in pos when inode pos is found
    int pos = 0;
    // loop thru inode list to find empty inode
    for (int j = 0; j < MAX_INODES; j++)
    {
        // find unused inode to use
        if (!inodes[j].val)
        {
            //set id
            inodes[j].id = _file_id;
            // save position in list
            pos=j;
            // set inode as valid
            inodes[j].val=true;
            break;
        }
    }

    // loop thru free blocks find block that is empty
    for (int f = 0; f < 512; f++)
    {
        // check if block is empty
        if (free_blocks[f] == '0')
        {
```

```
            // set block to used
            free_blocks[f] = '1';
            // set the block number in inode
            inodes[pos].block_num = f;
            return true;
        }
    }
    return false;

}
```

FileSystem.C: DeleteFile : In this function, I delete the file with its file id passed in as an arg. This is done by looking up all currently valid inodes and seeing if there is one with a matching file id when I find one that has a matching file ID I then set its block in the free blocks list to un-used, and then I set the inode to invalid.

```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("deleting file with id:"); Console::puti(_file_id);
    Console::puts("\n");
    /* First, check if the file exists. If not, throw an error.
       Then free all blocks that belong to the file and delete/invalidate
       (depending on your implementation of the inode list) the inode. */

    // loop thru inode list till you find one with the same file_id
    for(int i = 0; i<MAX_INODES; i++){
        // check if node is valid
        if (inodes[i].val)
        {
            // delete the inode and its blocks if it has the same id
            if (_file_id == inodes[i].id)
            {
                // set block to free
                free_blocks[inodes[i].block_num] = '0';
                // set inode to invalid
                inodes[i].val=false;
                return true;
            }
        }

    }
    return false;
}
```

FileSystem.C: get_disk : In this function, I return the disk object saved to the filesystem private variables, this comes in handy when I need to access the disk in the file class.

```
SimpleDisk * FileSystem::get_disk(){
    // return disk
    return disk;
}
```

File.C: constructor : In this function, I set all the private variables that are needed later to perform operations on the file object. I also read the file into the cache which all operations will take place on. Finally, I set the current position equal to 0.

```
File::File(FileSystem *_fs, int _id) {
```

```
    Console::puts("Opening file.\n");
    // set file system
    fs = _fs;
    // set inode
    inode = fs->LookupFile(_id);
    // set disk
    disk = fs->get_disk();
    // read disk to buffer
    disk->read(inode->block_num, block_cache);
    curr_poss = 0;

}
```

File.C: destructor : In this function, I write the cache back to the file before closing it.

```
File::~File() {
    Console::puts("Closing file.\n");
    // writing back to disk
    disk->write(inode->block_num, block_cache);
}
```

File.C: Read : In this function, I write n argument characters from the buffer into the cached file the buffer is passed in as a char* arguement to the buffer.

```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    // set counter for characters
    int count=0;
    for(int i = curr_poss; i<_n; i++)
    {
        // check end of file if it is then break;
        if (EoF())
        {
            break;
        }
        // read from buffer
        _buf[count] = block_cache[i];
        count++;
        curr_poss++;

    }
    return count;
}
```

File.C: Write : In this function, I write n argument characters from the buffer into the cached file the buffer is passed in as a char* arguement to the buffer.

```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    // set counter for characters
    int count=0;
    for(int i = curr_poss; i<_n; i++)
    {
        // check there is no space in the file then break;
        if (i == 512)
        {
```

```
            break;
        }
        // write a buffer
        block_cache[i] = _buf[count];
        count++;
        curr_poss++;

    }

    // update current poss if it is more than file length
    if (curr_poss > inode->file_length)
    {
        inode->file_length = curr_poss;
    }
    return count;
}
```

File.C: Reset : In this function, I reset the current position of the file.

```
void File::Reset() {
    Console::puts("resetting file\n");
    // reset current poss to 0
    curr_poss = 0;
}
```

File.C: EOF : In this function, I return a bool as to whether or not the current position has reached the end of the file.

```
bool File::EoF() {
    Console::puts("checking for EoF\n");
    // check if current poss is equal to file length
    if (curr_poss == inode->file_length)
    {
        return true;
    }
    return false;

}
```

Testing

I just used the tests given to prove that my implementation of the file system worked as it fully tested whether or not I was able to create files, read and write to them, and then delete them. It also helped me catch 2 bugs in my code. The first of these bugs was in the was in the create file function where it wasn't setting the inode as valid so the file could never be found by lookup file or delete file. The second bug was also in the create file function and it was basically failing to break out of the create file function after an inode was assigned and set valid. So considering that my program was able to run pretty much infinitely within the infinite loop set for the tests in kernel.C I believe that no further testing is needed as all of my functions are tested by the given test case.