MP7 - Vanilla File System

Chidambaram Ganesan
UIN: 434007396
CSCE 611: Operating Systems

Assigned Tasks:
Main: **Completed**
Bonus Option 1 - **Completed**
Bonus Option 2 - Did not attempt
Files Modified: file_system.C, file_system.H, file.C, file.H

System Design:

The file system is constructed within memory, utilizing the first two blocks on the disk for metadata storage while the remaining blocks are dedicated to holding file data. The initial block retains file information, and the second block tracks the occupancy status of memory blocks.

Three classes—FileSystem, Inode, and File—facilitate this file system's operation. The Inode class stores identifiers, sizes, and block locations for files, with a collection of these Inodes kept in the disk's block 0. The FileSystem class provides functionality for disk operations such as mounting, formatting, and managing files, including creating and deleting them. It keeps an in-memory representation of Inodes and block availability, which is loaded upon mounting a disk. File creation involves assigning an available Inode and a free block for the file, presuming the file's content occupies a single block. Conversely, deleting a file frees up its corresponding block and Inode.

The File class manages sequential read and write activities by keeping a local pointer to a specific file position. Opening a file uses its Inode to determine the block number, and the file's contents are cached for efficient access. The read and write processes adjust this local pointer, and the system ensures no writing past the block limit. When closing a file, the cached content is committed back to the disk.

Bonus 1 - DESIGN of an extension to the basic file system to allow for files that are up to 64kB long:
To handle larger files, a single block per file proves insufficient. For files as large as 64kB, we need 12 blocks.
Instead of a single block_number, we now maintain a list called block_list in both the inode and file structures, which sequentially records all block numbers where the file data is stored.

For example, to store a file of size 2kB, which requires 4 blocks, and if these blocks are numbered 23, 29, and 58, then the array block_list would contain [12, 14, 18, 20].

Modifications Required:
Inode Structure:

- Replace block_number with block_list, which will store the block numbers of all blocks where file data is stored.

File Class:

- Replace block_number with block_list to reflect the blocks used by the file.
- Read Operation: When reading from the file, if the end of the current block is reached, the operation should continue from the next block in the block_list unless it's the last block.
- Write Operation: If the end of a block is reached during writing and the file size does not exceed 64kB, a new free block should be assigned, added to the block_list, and then continue writing.
- End Of File (EOF): Return true if the end of the last block in block_list is reached; otherwise, return false.

FileSystem Class:

- Format: When formatting the disk, ensure all blocks listed in the block_list for each file are freed.
- CreateFile: Instead of storing a single block_number, the first free block assigned should be added to the block_list.
- DeleteFile: In addition to removing the file's data structures, all blocks recorded in the block_list should be freed.
- These adjustments will ensure the system can manage larger files efficiently, utilizing multiple blocks per file as needed.

Code Description

To compile the code, run the ‘make’ command, and to execute run the ‘make run’ command.

file_system.H

1. **Inode class**
    - Attributes:
        - long id - File "name"
        - unsigned int block_number
        - bool is_inode_free

        - int file_size
        - FileSystem* fs - Pointer to the FileSystem
    - Methods:
        - Inode() - Constructor

2. **FileSystem class**
    - Attributes:
        - unsigned int filesystem_size
        - unsigned int free_block_count
        - unsigned int inode_counter
        - unsigned char* free_blocks - Bitmap of free disk blocks
        - SimpleDisk* disk - Pointer to the associated disk
        - Inode* inodes - Array of Inodes
    - Methods:
        - FileSystem() - Constructor, initializes local data structures
        - ~FileSystem() - Destructor, unmounts the file system if mounted
        - bool Mount(SimpleDisk* _disk) - Associates the file system with a disk
        - static bool Format(SimpleDisk* _disk, unsigned int _size) - Formats the disk with a new, empty file system of given size
        - Inode* LookupFile(int _file_id) - Finds and returns the inode of a file with given id
        - bool CreateFile(int _file_id) - Creates a new file with the specified id
        - bool DeleteFile(int _file_id) - Deletes a file with the specified id

Constructor Inode::Inode()

```
/* Default constructor for Inode, initializes members to default values */
Inode::Inode() : fs(NULL), id(-1), is_inode_free(true), file_size(0) {}
```

Description: Initializes an inode instance by setting the default values for fs (filesystem pointer) to NULL, id to -1, is_inode_free to true, and file_size to 0. This sets up a newly instantiated inode as free and unassigned.

Constructor FileSystem::FileSystem()

```
FileSystem::FileSystem()
    : disk(nullptr),
      filesystem_size(0),
      free_block_count(SimpleDisk::BLOCK_SIZE / sizeof(unsigned char)),
      inode_counter(0) {

    free_blocks = new unsigned char[free_block_count];
    for (unsigned int i = 0; i < free_block_count; i++) {
        free_blocks[i] = 'F';
    }

    inodes = new Inode[MAX_INODES];

    Console::puts("File system: Free block list and inode array initialized.");
}
```

Description: Initializes a new instance of the file system by setting up an array of free blocks and inodes. It allocates memory for the free_blocks and inodes, and initializes all blocks to 'F' (free), indicating they are available for use.

Destructor FileSystem::~FileSystem()

```
/* Destructor for FileSystem class, writes data to disk before unmounting */
FileSystem::~FileSystem() {
    Console::puts("File system: Unmounting and saving data.");

    disk->write(0, free_blocks);
    unsigned char* tmp_inode_ref = reinterpret_cast<unsigned char*>(inodes);
    disk->write(1, tmp_inode_ref);

    Console::puts("File system: Successfully unmounted.");
    delete[] free_blocks;
    delete[] inodes;
}
```

Description: Safely unmounts the file system by writing the free block list and inode list back to the disk. It ensures that all file system data is saved before the object is destroyed, preventing data loss.

Function FileSystem::Mount(SimpleDisk* _disk)

```
bool FileSystem::Mount(SimpleDisk* _disk) {
    Console::puts("File system: Mounting from disk.");

    disk = _disk;
    disk->read(0, free_blocks);

    unsigned char* tmp_inode_ref = reinterpret_cast<unsigned char*>(inodes);
    disk->read(1, tmp_inode_ref);

    inode_counter = 0;
    for (unsigned int i = 0; i < MAX_INODES; i++) {
        if (!inodes[i].is_inode_free) {
            inode_counter++;
        }
    }

    Console::puts("File system: Mounted successfully.");
    return true;
}
```

Description: Attaches the file system to a specific disk. It reads the inode and free block information from the disk into the file system's memory, updating the inode counter to reflect the number of inodes currently in use.

Function FileSystem::Format(SimpleDisk* _disk, unsigned int _size)

```
/* Formats the disk, initializes inode list and free block list */
bool FileSystem::Format(SimpleDisk* _disk, unsigned int _size) {
    Console::puts("File system: Formatting disk.");

    unsigned int n_free_blks = _size / SimpleDisk::BLOCK_SIZE;
    unsigned char* free_blks_arr = new unsigned char[n_free_blks];
    free_blks_arr[0] = free_blks_arr[1] = 'U'; // Reserve blocks for metadata

    for (unsigned int i = 2; i < n_free_blks; i++) {
        free_blks_arr[i] = 'F';
    }

    _disk->write(0, free_blks_arr);

    Inode* tmp_inodes_ref = new Inode[MAX_INODES];
    unsigned char* tmp_inode_ref = reinterpret_cast<unsigned char*>(tmp_inodes_ref);
    _disk->write(1, tmp_inode_ref);

    delete[] free_blks_arr;
    delete[] tmp_inodes_ref;

    Console::puts("File system: Disk formatted successfully.");
    return true;
}
```

Description: Formats the disk by initializing it with a clean file system. This involves setting up an empty inode list and marking all blocks as free except for the first two, which are reserved for metadata.

Function FileSystem::LookupFile(int _file_id)

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("File system: Looking up file with ID = "); Console::puti(_file_id); Console::puts("\n");
    unsigned int i;
    for (i = 0; i < inode_counter; i++) {
        if (inodes[i].id == _file_id) {
            return &inodes[i];
        }
    }
    Console::puts("File system: File not found with ID = "); Console::puti(_file_id); Console::puts("\n");
    return NULL;
}
```

Description: Searches for a file with a specified identifier (_file_id). It scans the inode list and returns the inode if a match is found; otherwise, it returns NULL to indicate that the file does not exist.

Function FileSystem::CreateFile(int _file_id)

```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts("File system: Creating file with ID = ");
    Console::puti(_file_id);
    Console::puts("\n");

    for (unsigned int i = 0; i < inode_counter; i++) {
        if (inodes[i].id == _file_id) {
            Console::puts("File system: Error - File already exists with ID = ");
            Console::puti(_file_id);
            Console::puts("\n");
            assert(false);
        }
    }

    int free_inode_indx = -1;
    for (unsigned int i = 0; i < MAX_INODES; i++) {
        if (inodes[i].is_inode_free) {
            free_inode_indx = i;
            break;
        }
    }

    int free_blk_indx = -1;
    for (unsigned int i = 0; i < free_block_count; i++) {
        if (free_blocks[i] == 'F') {
            free_blk_indx = i;
            break;
        }
    }

    assert(free_inode_indx != -1 && free_blk_indx != -1);

    inodes[free_inode_indx].is_inode_free = false;
    inodes[free_inode_indx].fs = this;
    inodes[free_inode_indx].id = _file_id;
    inodes[free_inode_indx].block_number = free_blk_indx;

    free_blocks[free_blk_indx] = 'U';

    Console::puts("File system: File created successfully.");
    return true;
}
```

Description: Attempts to create a new file with a given identifier (_file_id). It first checks for any existing file with the same ID and, if none is found, allocates a free inode and a free block to the new file. It returns true on successful creation.

Function FileSystem::DeleteFile(int _file_id)

```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("File system: Deleting file with ID = "); Console::puti(_file_id); Console::puts("\n");
    bool file_exists = false;
    unsigned int indx = 0;
    while (indx < MAX_INODES) {
        if (inodes[indx].id == _file_id) {
            file_exists = true;
            break;
        }
        indx++;
    }
    assert(file_exists);
    int blk_number = inodes[indx].block_number;
    free_blocks[blk_number] = 'F';
    inodes[indx].is_inode_free = true;
    inodes[indx].file_size = 0;
    Console::puts("File system: File deleted successfully.");
    return true;
}
```

Description: Deletes a file specified by _file_id from the file system. It finds the inode for the file, frees its associated disk block, and marks the inode as free. This function ensures the resources are made available again for new files.

file.H

Attributes:

- FileSystem* file_system
- int file_id
- unsigned int block_number
- unsigned int inode_index
- unsigned int file_size
- unsigned int curr
- unsigned char block_cache[SimpleDisk::BLOCK_SIZE]

Functions

- File(FileSystem * _fs, int _id)
- ~File()
- int Read(unsigned int _n, char * _buf)
- int Write(unsigned int _n, const char * _buf)
- void Reset()
- bool EoF()

Constructor File::File(FileSystem* _fs, int _id)

```
File::File(FileSystem* _fs, int _id)
    : file_system(_fs), file_id(_id), curr(0) {

    Console::puts("Opening file.\n");

    bool file_block_found = false;
    for (unsigned int i = 0; i < file_system->MAX_INODES; ++i) {
        if (file_system->inodes[i].id == file_id) {
            inode_index = i;
            block_number = file_system->inodes[i].block_number;
            file_size = file_system->inodes[i].file_size;
            file_block_found = true;
            break;
        }
    }

    assert(file_block_found); // Ensure the file block is found
}
```

Initializes a file instance by locating the corresponding inode in the filesystem's inode array using the file ID. It sets up necessary metadata such as block number and file size, ensuring that the file exists with an assertion check.

Destructor File::~File()

```
File::~File() {
    Console::puts("Closing file.\n");
    // Write cached data to disk
    file_system->disk->write(block_number, block_cache);
    // Update inode in the inode list
    file_system->inodes[inode_index].file_size = file_size;
    unsigned char* tmp_inode_ref = reinterpret_cast<unsigned char*>(file_system->inodes);
    file_system->disk->write(1, tmp_inode_ref);
}
```

Writes back any changes from the cache to the disk and updates the inode list when the file is closed. This destructor ensures all data is saved and resources are cleanly released.

Function int File::Read(unsigned int _n, char* _buf)

```
/* Read data from file */
int File::Read(unsigned int _n, char* _buf) {
    Console::puts("Reading from file\n");
    unsigned int char_count = 0;
    for (unsigned int index = curr; index < file_size && char_count < _n; ++index) {
        _buf[char_count++] = block_cache[index];
    }
    curr += char_count;
    Console::puts("Reading from file complete.\n");
    return char_count;
}
```

Reads up to _n characters from the file into _buf, starting from the current position, and updates the position accordingly. It returns the number of characters read and ensures no read beyond the file size.

Function int File::Write(unsigned int _n, const char *_buf)

```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    int char_count = 0;
    int end_indx = curr + _n;
    while(curr < end_indx){
        if(curr == SimpleDisk::BLOCK_SIZE){
            Console::puts("EOF reached while writing.\n");
            break;
        }
        block_cache[curr++]=_buf[char_count++];
        file_size++;
    }

    Console::puts("writing to file complete.\n");
    return char_count;
}
```

Writes _n characters from _buf to the file starting at the current position, adjusting the file size if necessary. The function ensures that writing does not exceed the block size or file limits.

Function void File::Reset()

```
void File::Reset() {
    Console::puts("resetting file\n");
    curr=0;
}
```

Resets the current position in the file to zero, effectively setting the read/write pointer to the beginning for subsequent operations.

Function bool File::EoF()

```
bool File::EoF() {
    Console::puts("checking for EoF\n");
    return (curr<file_size);
}
```

Checks if the current position is at or beyond the end of the file, returning true if more data can be read, otherwise false.

Testing:

The functionality was verified by continuously creating, reading, writing, and deleting files within a file system using Kernel.C. After each action, the contents of the files were checked for accuracy. Two distinct strings were written alternately to test the files and verify thorough cleanup. Post-deletion, file lookup checks were conducted to confirm that the inodes were reset. Repeated tests ensured the block availability map was properly cleared after each operation.

```
QEMU
resetting file
Reading from file
Reading from file complete.
resetting file
Reading from file
Reading from file complete.
SUCCESS!!
Closing File 1 and File 2 again
Closing file.
Closing file.
Deleting File 1 and File 2
File system: Deleting file with ID = 1
File system: File deleted successfully.File system: Looking up file with ID =
File system: File not found with ID = 1
File system: Deleting file with ID = 2
File system: File deleted successfully.File system: Looking up file with ID =
File system: File not found with ID = 2
iteration done
EXCELLENT! Your File system seems to work correctly. Congratulations!!
```