MP7: Vanilla File System
Chonglin Zhang
UIN: 833003072
CSCE611: Operating Systems

Assigned Tasks

In this MP, the task is to develop a basic file system. This system will only allow linear access to files, and it utilizes a straightforward naming system where files are identified by unsigned integer values, excluding the use of complex multi-tier directory structures. Additionally, I have enhanced the fundamental file system design to accommodate files with a maximum size of 64kB. Will involves constructing this foundational file system.

Basic File System
Design
In this system, the management of files and file system operations is effectively split between two distinct classes: the FileSystem class and the File class. This separation fosters modularity and simplifies maintenance, with each class having a clear, dedicated role.

The FileSystem class acts as the steward of the system, overseeing the mapping between the file namespace and the actual files. It is responsible for allocating files, managing available space, and addressing file system-related issues. By proficiently managing the directory structure, it ensures quick and easy access to the correct files when needed.

Conversely, the File class is dedicated to the manipulation of individual files. It offers a full suite of interfaces for file operations such as reading, writing, and resetting the file pointer. This encapsulation of file operations in the File class makes these actions intuitive and straightforward for users.

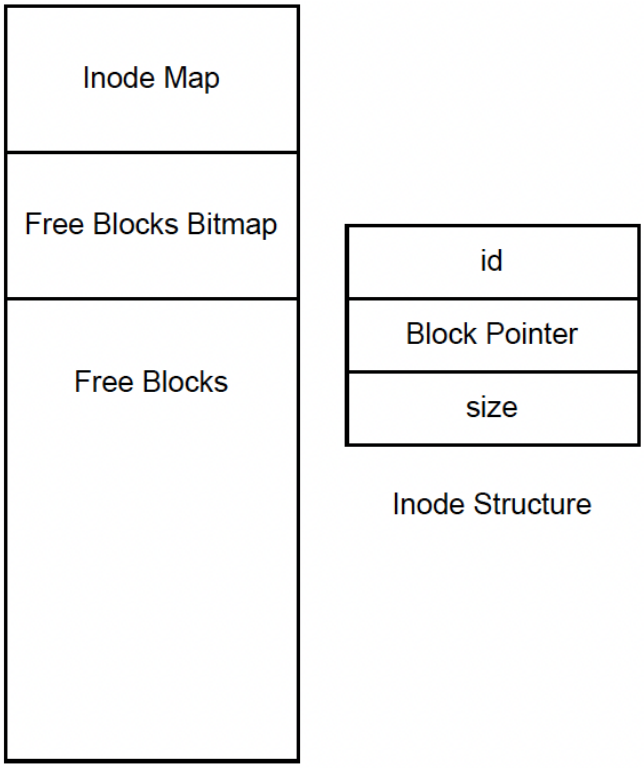
The disk setup includes two essential reserved blocks. The first reserved block is allocated for the inode map, which directly stores inodes, diverging from the conventional use of a free inode bitmap. This design choice is aimed at keeping the file system structure simple and direct.

The second reserved block is dedicated to the free block bitmap, providing a visual representation of disk space usage. It indicates which blocks are occupied and which are free, facilitating efficient space management and allocation.

The rest of the disk space, starting from the third block, consists of data blocks where actual file data is stored. Each block is 512 bytes, a size chosen for its balance of space efficiency and manageability. The organization of these blocks and the methods of reading and writing them are crucial for the overall performance of the file system.

The initialization process of the file system starts by configuring the initial three reserved blocks. Subsequently, within the first of these blocks, an array of inodes is stored. It will look like the image below.

This is the file system and inode structure



During the initialization of the file system, the second block is set up with an array of free blocks. This block acts as a map, providing a clear overview of disk space usage by marking which blocks are available and which are already in use.

To read a file, the file system begins by navigating through the inode list to find the inode that corresponds to the specific file id being requested. Once this inode is located, it reveals the necessary details to identify the exact block where the file's data is stored. The system then retrieves the data from this block, making it accessible for the user or application.

For a write request, the file system employs a systematic method to ensure efficient storage. The process starts with finding an unoccupied block in the array within the second reserved block. Upon locating such a block, the system marks it as allocated, indicating that the space is now in use. Following this, the file system searches for an available inode in the inode list. It then updates the chosen inode, altering the file id field to match the id of the new file and adjusting the block pointer to direct to the newly allocated block. After these updates, the file system writes the data and metadata back to the disk, thus completing the write operation.

Implementation

file_system.C: Mount() : The function plays a crucial role in setting up the file system for operation, primarily by linking it to a disk. This is accomplished initially by saving a reference to the disk within a private member variable. Following this, the function populates the inodes array and the list of free blocks by extracting the relevant data from the first two blocks on the disk. These steps provide the file system with essential information needed for the management of files and the allocation of free blocks on the disk.

```
bool FileSystem::Mount(SimpleDisk *_disk) {
    if (!_disk) {
        Console::puts("Failed to mount: No disk provided.\n");
        return false;
    }

    Console::puts("Mounting file system from disk\n");
    disk = _disk;
    disk->read(0, reinterpret_cast<unsigned char *>(inodes));
    disk->read(1, free_blocks);

    return true;
}
```

file_system.C: Format() : This function plays a pivotal role in preparing a disk for utilization with the file system. It begins by ensuring the specified size of the disk is within acceptable parameters. Following this validation, the function proceeds to clear the inode list. This is done by writing an empty inode structure into each slot of the inode block, effectively resetting all inodes to their initial state. Next, the function initializes the free block list. In this step, it designates all blocks as free, with the exception of the first two reserved blocks – the inode list block and the free block list block. The function then calculates the total number of blocks required for the given file system size. Any blocks exceeding this calculated number are marked as used to prevent their allocation. The final step involves updating the disk with the newly configured free block list. This action concludes the disk formatting process, rendering the disk ready and optimized for subsequent file system operations.

```
bool FileSystem::Format(SimpleDisk *_disk, unsigned int _size) {
    if (_size < 2 || _size > _disk->size()) {
        return false;
    }
    unsigned char* block_node_i = new unsigned char[SimpleDisk::BLOCK_SIZE];
    Inode list_empty_node{};
    list_empty_node.id = -1;
    for (unsigned int i = 0; i < MAX_N; ++i) {
        unsigned int offset = i * sizeof(Inode);
        for (unsigned int j = 0; j < sizeof(Inode); ++j) {
            block_node_i[offset + j] = reinterpret_cast<unsigned char*>(&list_empty_node)[j];
        }
    }
    _disk->write(0, block_node_i);
    delete[] block_node_i;

    unsigned char* block_free_list = new unsigned char[SimpleDisk::BLOCK_SIZE];
    for (unsigned int i = 0; i < SimpleDisk::BLOCK_SIZE; ++i) {
        block_free_list[i] = (i < 2) ? 0x00 : 0xFF;
    }

    unsigned int need_size = _size * 1024;

    unsigned int num_blocks = need_size / SimpleDisk::BLOCK_SIZE;
    for (unsigned int i = num_blocks; i < SimpleDisk::BLOCK_SIZE; ++i) {
        block_free_list[i] = 0x00;
    }

    _disk->write(1, block_free_list);
    delete[] block_free_list;

    return true;
}
```

file_system.C: LookupFile() : This function is specifically engineered to locate and return a pointer to the inode that matches a specified file id. To achieve this, it systematically iterates through the array of inodes, comparing the id of each inode with the provided file id. When it finds an inode whose id matches the given file id, the function returns a pointer to that particular inode. This capability is essential in the file system for efficiently identifying and accessing specific files based on their unique identifiers.

```
Inode *FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = ");
    Console::puti(_file_id);
    Console::puts("\n");
    unsigned int i = 0;
    while (i < MAX_N) {
          if (inodes[i].id == _file_id) {
              return &inodes[i];
          }
          ++i;
    }
    return nullptr;
}
```

file_system.C: CreateFile() : This function is tasked with creating a new file in the file system using a specified file id. The process begins with a check for the existence of a file with the same id, using the LookupFile() function. If a file with the given id is already present, the function halts and returns an error message. In the case where no file with the provided id exists, the function proceeds to locate a free inode. This is accomplished through the GetFreeInode() function. Upon successfully finding a free inode, the function then initializes this inode with the given file id. It sets the size of the file to zero and integrates this inode into the current file system. Further, the function initializes all direct and indirect block pointers associated with this inode to zero. This step is crucial to ensure that the inode correctly references the file's location in the system. Once these initializations are complete, the updated inode list is saved back to the disk, finalizing the creation of the new file.

```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts("creating file with id:");
    Console::puti(_file_id);
    Console::puts("\n");
    if (LookupFile(_file_id) != nullptr) {
        Console::puts("Failure: File already exists.");
        return false;
    }

    // Find a free inode index
    short index_free_node = GetFreeInode();
    if (index_free_node < 0) {
        Console::puts("Failure: All inodes are currently in use.");
        return false;
    }

    // Initialize the inode
    Inode *inode = &inodes[index_free_node];
    inode->id = _file_id;
    inode->size = 0;
    inode->fs = this;

    // block pointers Initialized direct and indirect
    for (auto &block : inode->direct_blocks) {
         block = 0;
    }
    inode->indirect_block = 0;

    // Save the inode list
    EntireInode();

    return true;
}
```

file_system.C: DeleteFile() : The function's purpose is to eliminate a file, identified by a given file id, from the file system. The procedure begins with locating the inode that corresponds to the specified file id. If this inode is successfully found, the function initiates the removal process. The first step in this process involves releasing all direct and indirect blocks associated with the inode. This is achieved by marking these blocks as available in the free blocks list, thereby freeing up the space they occupied. Subsequently, the function resets the size of the inode to zero and updates the inode with these changes. The next phase is to officially remove the file's record from the inode list. This is done by changing the inode's id to -1, which signifies that the inode is no longer linked to any active file. After making this adjustment, the updated inode list is saved back to the disk. Once all these steps are successfully completed, the function concludes its operation by returning true. This return value confirms that the file has been successfully deleted from the file system.

```
bool FileSystem::DeleteFile(int _file_id) {
    Inode *inode = LookupFile(_file_id);

    if (inode == nullptr) {
        return false;
    }

    for (unsigned int &direct_block : inode->direct_blocks) {
        if (direct_block == 0) {
            continue;
        }

        free_blocks[direct_block] = 0xFF;
        direct_block = 0;
    }

    if (inode->indirect_block != 0) {
        unsigned char indirect_block_cache[SimpleDisk::BLOCK_SIZE];
        //ReadBlock(inode->indirect_block, indirect_block_cache);
        disk->read(inode->indirect_block, indirect_block_cache);

        for (int i = 0; i < ippb; ++i) {
            unsigned int data_block;
            memcpy(&data_block, indirect_block_cache + i * sizeof(unsigned int), sizeof(unsigned int));

            if (data_block == 0) {
                continue;
            }

            free_blocks[data_block] = 0xFF;
        }

        free_blocks[inode->indirect_block] = 0xFF;
        inode->indirect_block = 0;
    }

    inode->size = 0;
    StoreInode(inode);
    inode->id = -1;
    EntireInode();

    return true;
}
```

file.C: constructor() : The function initiates its process by first capturing the reference to the provided file system. Utilizing this reference, it proceeds to search for the inode that matches the specified file id. After locating the appropriate inode, the function sets the current position pointer to 0. This action signifies that the starting point for any file operations will be at the beginning of the file. Subsequently, the function reads the first block of the file from the disk, transferring this block into the block cache. This step is crucial as it primes the block for any forthcoming read or write operations on the file. By carrying out these steps, the constructor ensures that a File object is not only correctly initialized but is also immediately prepared for operational use following its creation. This process streamlines the setup for handling files within the system, facilitating efficient and immediate access to file data.

```
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file with id = ");
    Console::puti(_id);
    Console::puts("\n");
    fs = _fs;
    position = 0;
    block_cache = new unsigned char[SimpleDisk::BLOCK_SIZE];
    edit_block_cache = false;
    inode = _fs->LookupFile(_id);

    if (!_fs) {
        Console::puts("Error: null pointer exist\n");
        assert(false);
     }

    if (!inode) {
        Console::puts("Error: Need to find File\n");
        return;
    }
}
```

file.C: Read() : This function is tailored for reading a specified number of bytes from a file into a given buffer. The operation unfolds through several steps:
1. Identification of Block Index and Offset: The function begins by determining the block index and the offset within that block where the reading should start, and it calculates the number of bytes to be read in each iteration.

2. Reading from Direct Blocks: If the block index is within the range of direct blocks, the function reads directly from these blocks.

3. Handling Indirect Blocks: When the block index falls within the indirect blocks range, the function reads the corresponding indirect block to extract the actual data block number and then reads from this data block.

4. Storing Data in Block Cache: Whether reading from direct or indirect blocks, the function stores the read data in a block cache. This intermediate storage is crucial for processing the data before moving it to the buffer.

5. Continuous Reading Process: The reading process continues, with the function iterating through the blocks and storing the data in the block cache until it has read the required number of bytes or reached the end of the file.

6. Transferring Data to Buffer: After the necessary data is read and stored in the block cache, it is then copied into the provided buffer.

7. Return Value: The function concludes by returning the total number of bytes read from the file.

This approach ensures efficient reading of files, accommodating both direct and indirect block access, and effectively manages the transfer of data from the file system to the application layer through the buffer.

```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("Reading file with id = ");Console::puti(inode->id);Console::puts("\n");

    int bytesRead = 0;
    while (_n > 0 && !EoF()) {
        bytesRead += readNextBlock(_n, _buf + bytesRead);
    }

    Console::puts("Finished reading ");Console::puti(bytesRead);Console::puts(" bytes \n");

    return bytesRead;
}

int File::readNextBlock(unsigned int &n, char *destination) {
    unsigned int blockIndex = position / SimpleDisk::BLOCK_SIZE;
    unsigned int offset = position % SimpleDisk::BLOCK_SIZE;
    int bytesToRead = (n < SimpleDisk::BLOCK_SIZE - offset) ? n : SimpleDisk::BLOCK_SIZE - offset;

    loadAndReadBlock(blockIndex, offset, bytesToRead, destination);

    position += bytesToRead;
    n -= bytesToRead;
    return bytesToRead;
}

void File::loadAndReadBlock(unsigned int blockIndex, unsigned int offset, int bytesToRead, char *destination) {
    if (blockIndex < NUM_DIRECT_BLOCKS) {
        fs->ReadBlock(inode->direct_blocks[blockIndex], block_cache);
    } else {
        loadIndirectBlock(blockIndex);
    }
    memcpy(destination, block_cache + offset, bytesToRead);
    edit_block_cache = false;
}

void File::loadIndirectBlock(unsigned int blockIndex) {
    unsigned int indirectIndex = (blockIndex - NUM_DIRECT_BLOCKS) / ippb;
    unsigned int indirectOffset = (blockIndex - NUM_DIRECT_BLOCKS) % ippb;

    auto indirectBlock = new unsigned char[SimpleDisk::BLOCK_SIZE];
    fs->ReadBlock(inode->indirect_blocks[indirectIndex], indirectBlock);

    unsigned long dataBlockNumber;
    memcpy(&dataBlockNumber, indirectBlock + indirectOffset * sizeof(unsigned long), sizeof(unsigned long));
    delete[] indirectBlock;

    fs->ReadBlock(dataBlockNumber, block_cache);
}
```

file.C: Write() : The function is designed to write a specific number of bytes from a buffer into a file. The writing process involves several key steps:

1. Calculating Block Index and Offset: It starts by calculating the block index and the offset within the block to identify the exact location for writing.

2. Tracking Remaining Bytes: The function keeps track of the remaining bytes that need to be written and iterates over this amount until all bytes are written or the file reaches its size limit.

3. Handling Direct and Indirect Blocks: It is capable of writing to both direct and indirect blocks. If an indirect block is needed but does not exist, the function allocates one. Similarly, a new block is allocated if the target block for writing does not exist.

4. Reading and Writing to Block Cache: The function reads the relevant block into a block cache, then copies the data from the buffer to the block cache, and finally writes the block cache back to the block.

5. Updating File Size in the Inode: If the write operation extends the file size, the function updates this new size in the inode.

6. Returning Bytes Written: At the conclusion of the process, the function returns the number of bytes that have been successfully written to the file.

This approach ensures that the function efficiently handles the writing of data to files, managing the allocation of new blocks when necessary and updating the file size in the inode. The process is designed to be robust and accommodate various scenarios during the file writing operation. (Helper functions are in file.C which below int File::write())

```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("Writing file with id = ");Console::puti(inode->id);Console::puts("\n");
    unsigned int bytes_written = 0, remaining_bytes = _n;
    unsigned int block_index, offset_in_block, bytes_to_write;
    auto *indirect_block = new unsigned int[ippb];

    while (remaining_bytes > 0) {
        CalculateBlockPosition(position, block_index, offset_in_block);

        bytes_to_write = MIN(SimpleDisk::BLOCK_SIZE - offset_in_block, remaining_bytes);

        if (HandleBlockSizeLimit(block_index)) break;

        unsigned int block_to_read = DetermineBlockToRead(block_index, indirect_block);
        if (block_to_read == 0) {
            block_to_read = AllocateNewBlock(block_index, indirect_block);
            if (block_to_read == 0) break;
        }

        ProcessWrite(block_to_read, offset_in_block, _buf, bytes_written, bytes_to_write);

        UpdatePosition(bytes_to_write, remaining_bytes, bytes_written);
    }

    delete[] indirect_block;
    Console::puts("Finished writing ");Console::puti(bytes_written);Console::puts(" bytes\n");
    return bytes_written;
}
```

Testing

Below the image is test for all implementations which able to see the kernel and output console.