MP7: Vanilla File System

Vinay Balamurali
UIN: 936002032
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus 1: Completed.
Bonus 2: Completed.

System Design

This machine problem aims to implement a Simple File System that supports files up to 64 KB in size and supports sequential access only. Additionally, only single directory files are supported.

Main

The class `FileSystem` is responsible for maintaining the INODE list, as well as the FREE BLOCK list. The first two data blocks of the File System are reserved to store the previously mentioned data structures and in that order. The class `File` is tasked with performing read/write operations, thus keeping the implementation of the data blocks hidden to the end user.

Bonus 1 and 2: Design and implementation of an extension to the basic file system to allow for files up to 64KB long

In the original file system, the file size is restricted to 1 block, that is, 512 bytes. The block number would be stored in the `Inode` list for that file. To extend the file system, to accommodate larger files, we need to implement a mechanism to keep track of all the data blocks that the file occupies. This can be done using the **Indexed Allocation** scheme. In Indexed allocation, the Inode only has a reference to a data block that contains an array of pointers to each data block. In theory, this mechanism can be utilized for non-sequential access, but in this machine problem- free blocks are allocated sequentially. Every time a file has to be read/written, the class `File` looks up the index table, which itself occupies up to 1 data block. The table is traversed sequentially to access the file. The maximum elements the table can hold is 512 (limited by the block size for now). The table is stored as an `unsigned char` where the value represents the physical data block where the data are stored for the file. Unoccupied data blocks are represented using `255`. So numbers 0 to 254 are actually usable. For a file of size 64 KB, you would require 128 blocks. So this implementation can in theory hold larger files but is restricted by a macro called `MAX_BLOCKS`. Details of the implementation are mentioned in the Code Desscription section.

Code Description

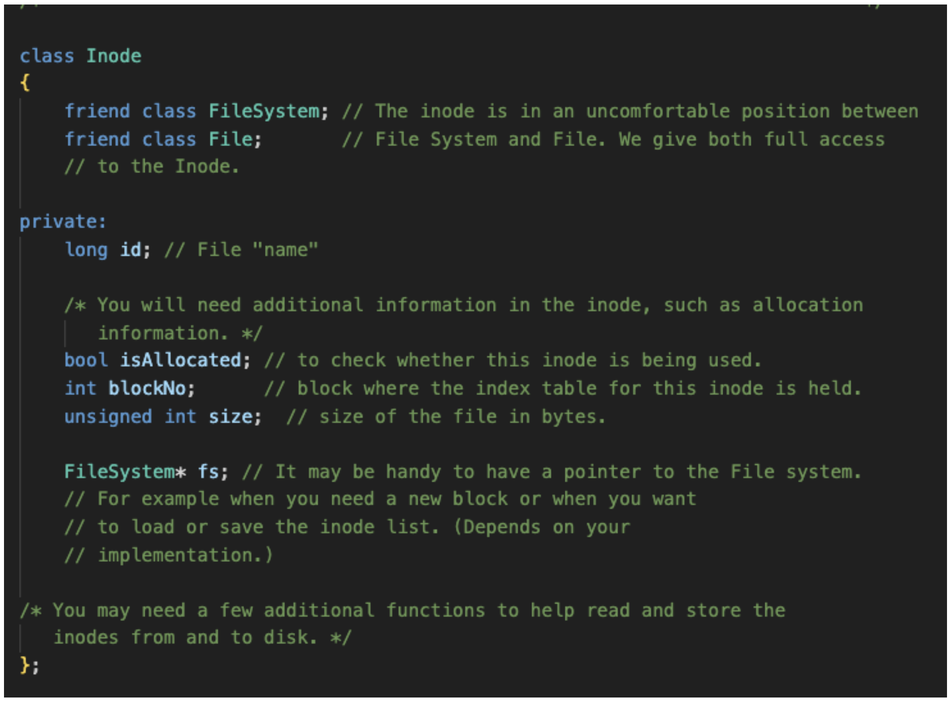
0.1 Files Added/Modified

- file_system.H
- file_system.C
- file.H
- file.C
- kernel.C

file_system.H : This header file contains method declarations for class `Inode` and class `FileSystem`.

Implementation of Class `Inode`:

- `long id` : Unique identifier for the file.
- `bool isAllocated`: Indicates if the inode is currently in use.
- `int blockNo`: The block number where the file's index table resides.
- `unsigned int size`: Size of the file in bytes.
- `FileSystem* fs`: A pointer to the file system for accessing global operations.



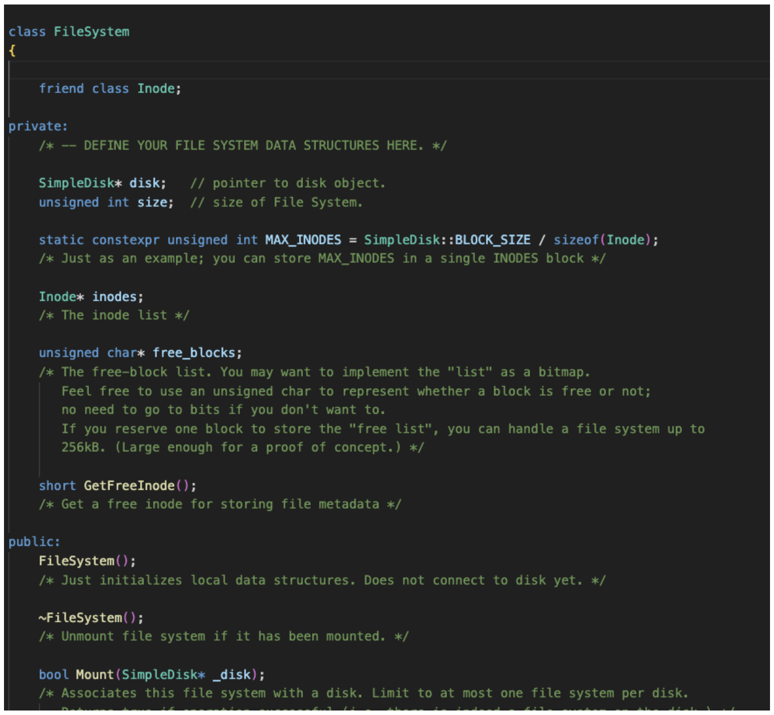
```
class Inode
{
    friend class FileSystem; // The inode is in an uncomfortable position between
    friend class File;       // File System and File. We give both full access
    // to the Inode.

private:
    long id; // File "name"

    /* You will need additional information in the inode, such as allocation
       information. */
    bool isAllocated; // to check whether this inode is being used.
    int blockNo;      // block where the index table for this inode is held.
    unsigned int size;  // size of the file in bytes.

    FileSystem* fs; // It may be handy to have a pointer to the File system.
    // For example when you need a new block or when you want
    // to load or save the inode list. (Depends on your
    // implementation.)

/* You may need a few additional functions to help read and store the
   inodes from and to disk. */
};
```

Figure 1: `class Inode`

The class `FileSystem` manages the core operations of the file system, including file creation, deletion, and data management. Added/Modified components include:

- `Inode* inodes`: List of inodes, managing metadata for all files in the system.
- `void readBlockFromDisk(int blockNo, unsigned char *buffer)`: Reads data from a specific block into the provided buffer.

- `void writeBlockToDisk(int blockNo, unsigned char *buffer)`: Writes data from the buffer into a specific block on the disk.
- `void writeInodeListToDisk()`: Persists the inode list to the disk for storage and recovery purposes.
- `int GetFreeBlock()`: Allocates and returns a free block number, marking it as used in the free block list.
- `short GetFreeInode()`: Retrieves a free inode for storing file metadata. This method ensures efficient management of the inode list by tracking usage.

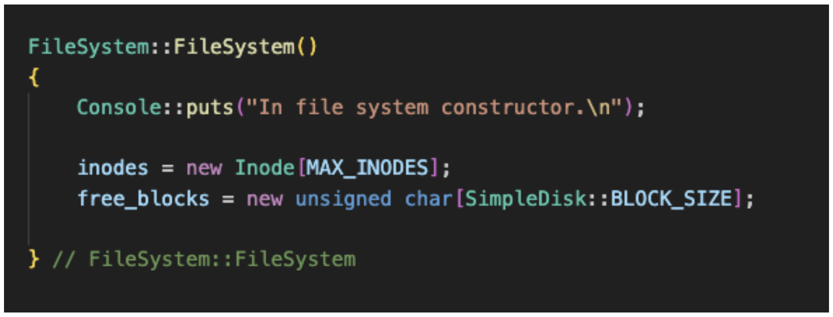


```
class FileSystem
{

    friend class Inode;

private:
    /* -- DEFINE YOUR FILE SYSTEM DATA STRUCTURES HERE. */

    SimpleDisk* disk;   // pointer to disk object.
    unsigned int size;  // size of File System.

    static constexpr unsigned int MAX_INODES = SimpleDisk::BLOCK_SIZE / sizeof(Inode);
    /* Just as an example; you can store MAX_INODES in a single INODES block */

    Inode* inodes;
    /* The inode list */

    unsigned char* free_blocks;
    /* The free-block list. You may want to implement the "list" as a bitmap.
       Feel free to use an unsigned char to represent whether a block is free or not;
       no need to go to bits if you don't want to.
       If you reserve one block to store the "free list", you can handle a file system up to
       256kB. (Large enough for a proof of concept.) */

    short GetFreeInode();
    /* Get a free inode for storing file metadata */

public:
    FileSystem();
    /* Just initializes local data structures. Does not connect to disk yet. */

    ~FileSystem();
    /* Unmount file system if it has been mounted. */

    bool Mount(SimpleDisk* _disk);
    /* Associates this file system with a disk. Limit to at most one file system per disk.
```

Figure 2: `class FileSystem`

file_system.C: FileSystem::FileSystem() : This is the constructor of this class. `FileSystem::FileSystem()` initializes the FileSystem object by allocating memory for the inode list and free block list.

- Allocates memory for `inodes` and `free_blocks` arrays.
- Logs the initialization process to the console.

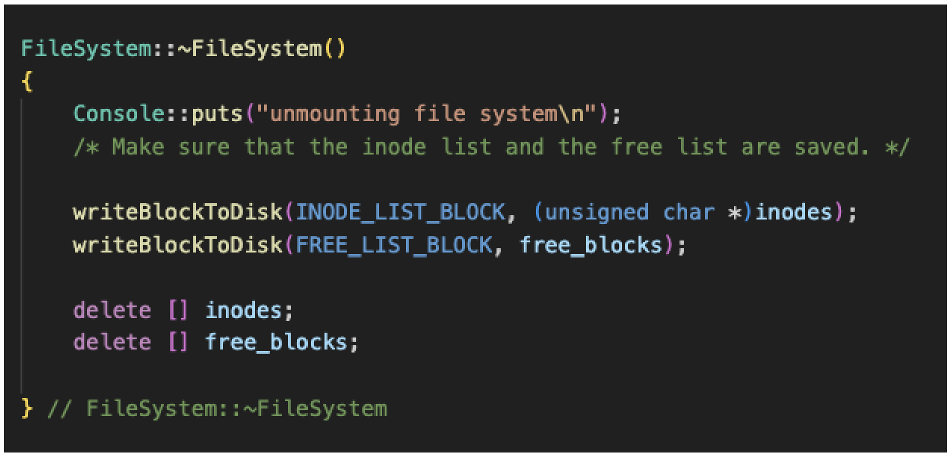


```
FileSystem::FileSystem()
{
    Console::puts("In file system constructor.\n");

    inodes = new Inode[MAX_INODES];
    free_blocks = new unsigned char[SimpleDisk::BLOCK_SIZE];

} // FileSystem::FileSystem
```

Figure 3: **class FileSystem** constructor

file_system.C: FileSystem::˜FileSystem() : This is the destructor of this class. It ensures the file system's state is saved before de allocating resources.

- Writes the inode list and free block list back to disk.
- Deallocates memory used by `inodes` and `free_blocks`.
- Logs the unmounting process to the console.

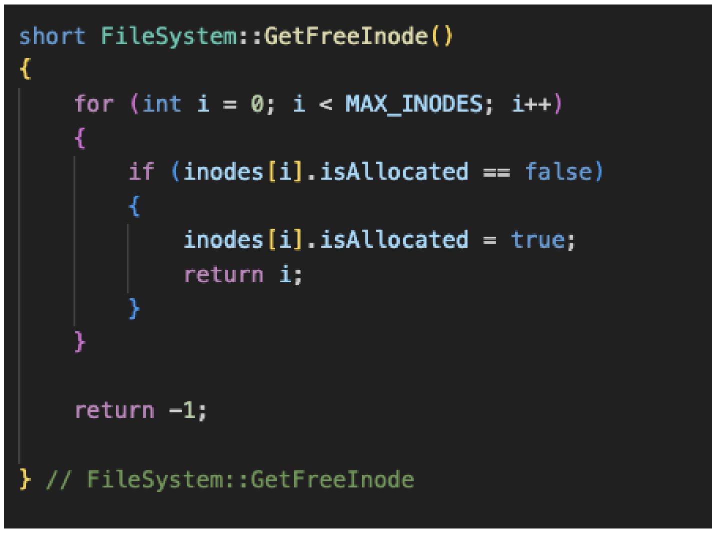


```
FileSystem::~FileSystem()
{
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */

    writeBlockToDisk(INODE_LIST_BLOCK, (unsigned char *)inodes);
    writeBlockToDisk(FREE_LIST_BLOCK, free_blocks);

    delete [] inodes;
    delete [] free_blocks;

} // FileSystem::~FileSystem
```

Figure 4: **class FileSystem** destructor

file_system.C: FileSystem::GetFreeInode() : This method searches the inode list for a free inode and marks it as allocated.

- Iterates through `inodes` to find an unallocated inode.
- Returns the index of the free inode or `-1` if none are available.

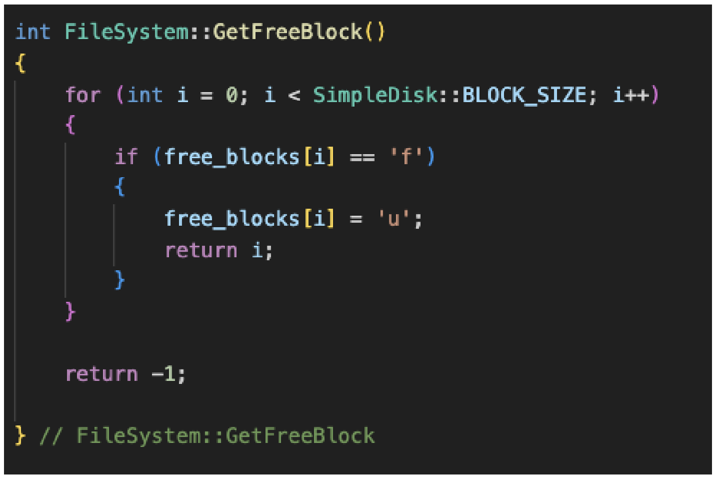


```
short FileSystem::GetFreeInode()
{
    for (int i = 0; i < MAX_INODES; i++)
    {
        if (inodes[i].isAllocated == false)
        {
            inodes[i].isAllocated = true;
            return i;
        }
    }

    return -1;

} // FileSystem::GetFreeInode
```

Figure 5: **GetFreeInode** method

file_system.C: FileSystem::GetFreeBlock() : This method retrieves a free block from the free block list.

- Searches `free_blocks` for an entry marked as free (`'f'`).
- Marks the block as used (`'u'`) and returns its index.
- Returns `-1` if no blocks are free.

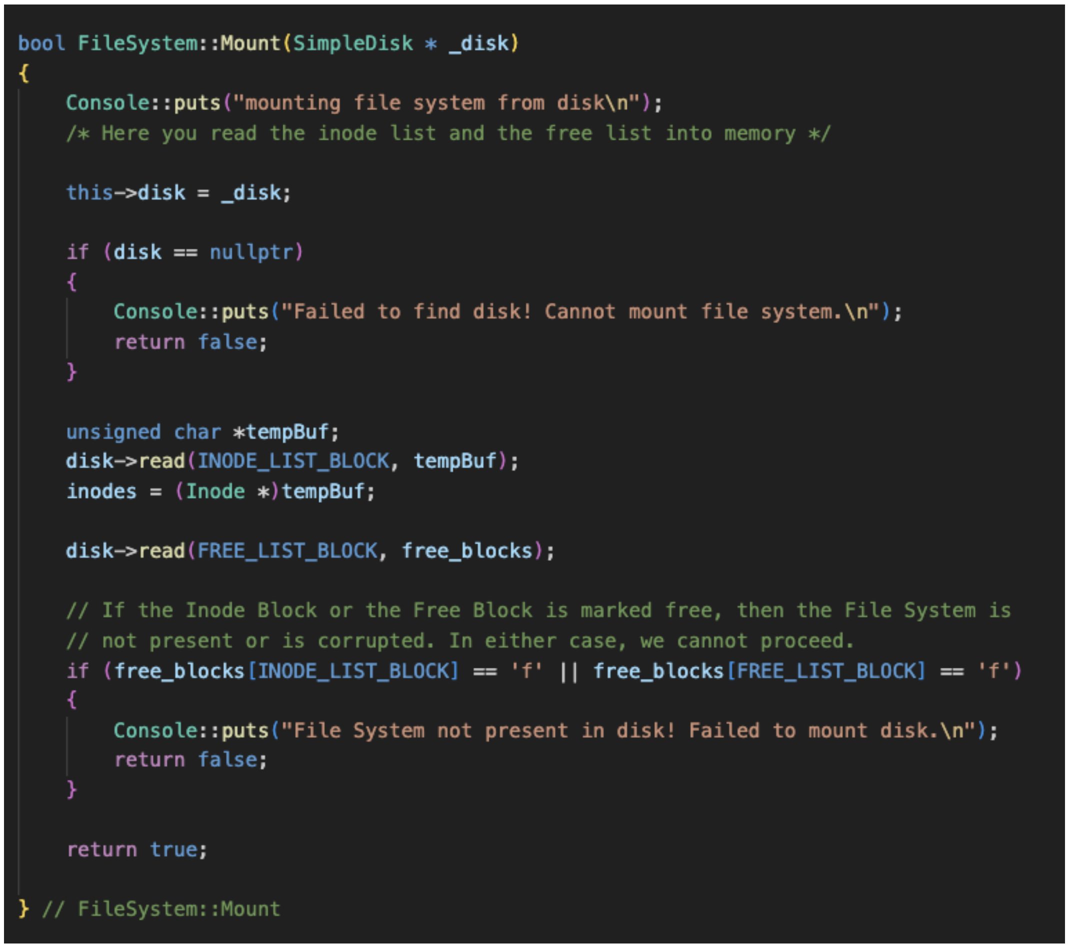


```
int FileSystem::GetFreeBlock()
{
    for (int i = 0; i < SimpleDisk::BLOCK_SIZE; i++)
    {
        if (free_blocks[i] == 'f')
        {
            free_blocks[i] = 'u';
            return i;
        }
    }

    return -1;

} // FileSystem::GetFreeBlock
```

Figure 6: **GetFreeBlock** method

file_system.C: FileSystem::Mount(SimpleDisk * _disk) : This method associates the file system with a disk and loads its state into memory.

- Reads the inode list and free block list from disk.
- Verifies that essential blocks are not marked as free, ensuring a valid file system is present.
- Logs success or failure to the console.

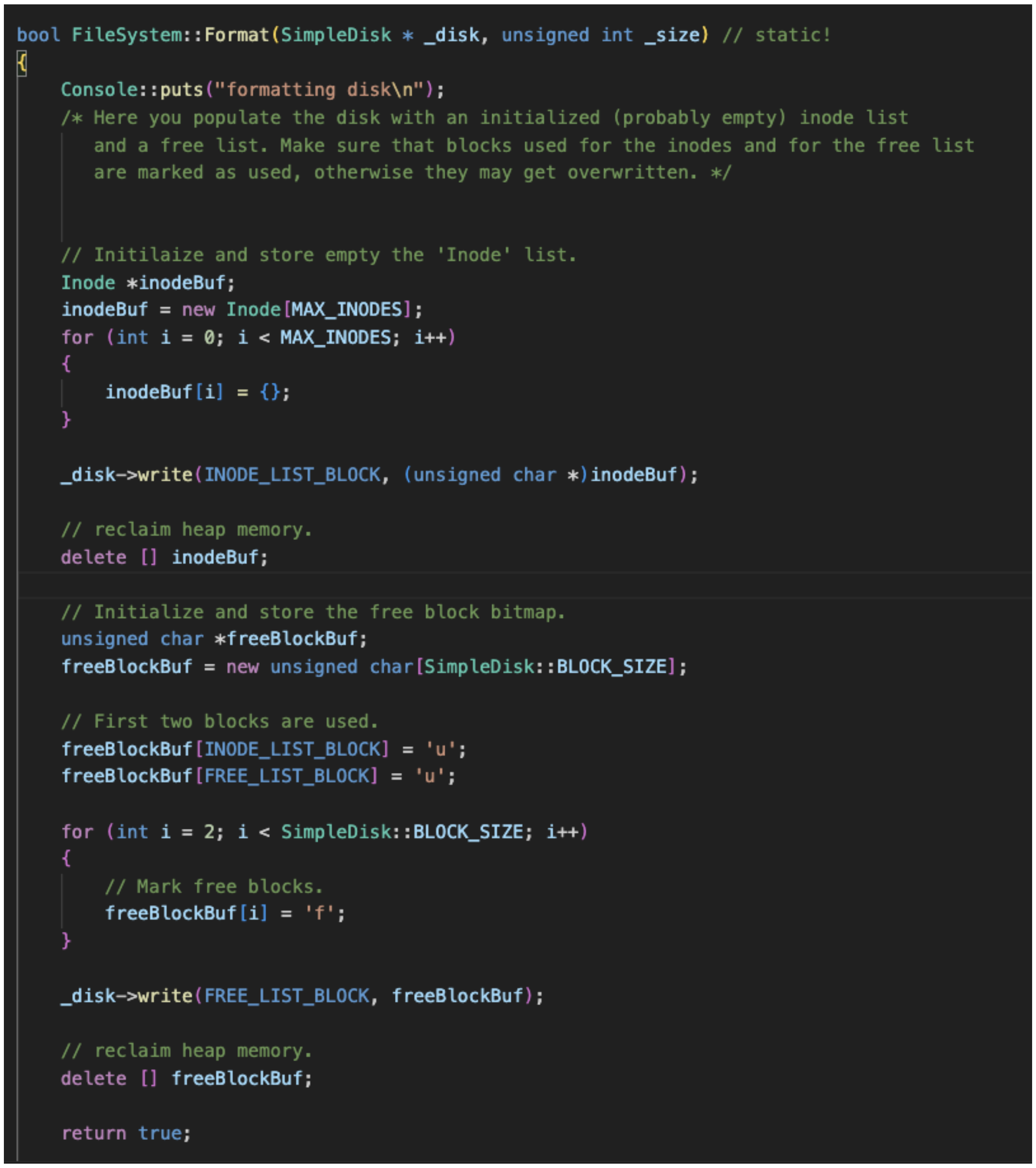


```
bool FileSystem::Mount(SimpleDisk * _disk)
{
    Console::puts("mounting file system from disk\n");
    /* Here you read the inode list and the free list into memory */

    this->disk = _disk;

    if (disk == nullptr)
    {
        Console::puts("Failed to find disk! Cannot mount file system.\n");
        return false;
    }

    unsigned char *tempBuf;
    disk->read(INODE_LIST_BLOCK, tempBuf);
    inodes = (Inode *)tempBuf;

    disk->read(FREE_LIST_BLOCK, free_blocks);

    // If the Inode Block or the Free Block is marked free, then the File System is
    // not present or is corrupted. In either case, we cannot proceed.
    if (free_blocks[INODE_LIST_BLOCK] == 'f' || free_blocks[FREE_LIST_BLOCK] == 'f')
    {
        Console::puts("File System not present in disk! Failed to mount disk.\n");
        return false;
    }

    return true;

} // FileSystem::Mount
```

Figure 7: **Mount** method

file_system.C: FileSystem::Format(SimpleDisk * _disk, unsigned int _size) : This method initializes the disk with an empty file system.

- Creates an empty inode list and writes it to disk.
- Initializes the free block bitmap, marking reserved blocks as used.
- Logs the formatting process to the console.

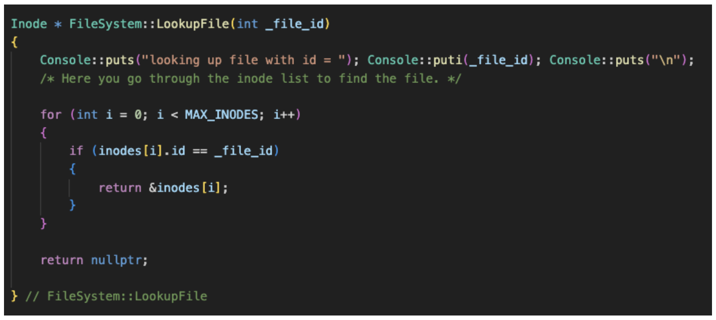


```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) // static!
{
    Console::puts("formatting disk\n");
    /* Here you populate the disk with an initialized (probably empty) inode list
       and a free list. Make sure that blocks used for the inodes and for the free list
       are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


    // Initilaize and store empty the 'Inode' list.
    Inode *inodeBuf;
    inodeBuf = new Inode[MAX_INODES];
    for (int i = 0; i < MAX_INODES; i++)
    {
        inodeBuf[i] = {};
    }

    _disk->write(INODE_LIST_BLOCK, (unsigned char *)inodeBuf);

    // reclaim heap memory.
    delete [] inodeBuf;

    // Initialize and store the free block bitmap.
    unsigned char *freeBlockBuf;
    freeBlockBuf = new unsigned char[SimpleDisk::BLOCK_SIZE];

    // First two blocks are used.
    freeBlockBuf[INODE_LIST_BLOCK] = 'u';
    freeBlockBuf[FREE_LIST_BLOCK] = 'u';

    for (int i = 2; i < SimpleDisk::BLOCK_SIZE; i++)
    {
        // Mark free blocks.
        freeBlockBuf[i] = 'f';
    }

    _disk->write(FREE_LIST_BLOCK, freeBlockBuf);

    // reclaim heap memory.
    delete [] freeBlockBuf;

    return true;
```

Figure 8: **Format** method

file_system.C: FileSystem::LookupFile(int _file_id) : This method searches for a file by its ID in the inode list.

- Iterates through `inodes` to locate a matching file ID.
- Returns a pointer to the inode if found, or `nullptr` if not.

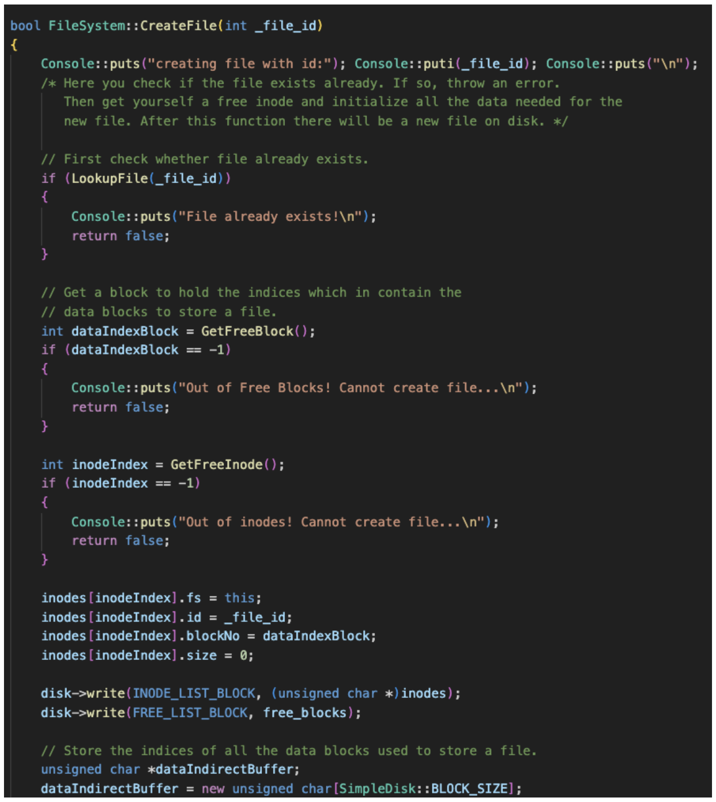


```
Inode * FileSystem::LookupFile(int _file_id)
{
    Console::puts("looking up file with id = "); Console::puti(_file_id); Console::puts("\n");
    /* Here you go through the inode list to find the file. */

    for (int i = 0; i < MAX_INODES; i++)
    {
        if (inodes[i].id == _file_id)
        {
            return &inodes[i];
        }
    }

    return nullptr;

} // FileSystem::LookupFile
```

Figure 9: **LookupFile** method

file_system.C: FileSystem::CreateFile(int _file_id) : This method creates a new file in the file system.

- Checks if a file with the given ID already exists.
- Allocates a free inode and a free block for file metadata.
- Initializes the inode with file metadata and writes updated structures to disk.

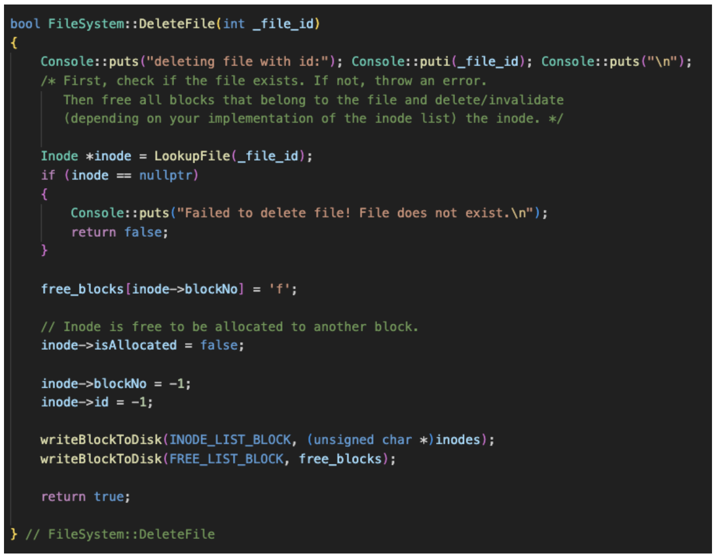


```
bool FileSystem::CreateFile(int _file_id)
{
    Console::puts("creating file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */

    // First check whether file already exists.
    if (LookupFile(_file_id))
    {
        Console::puts("File already exists!\n");
        return false;
    }

    // Get a block to hold the indices which in contain the
    // data blocks to store a file.
    int dataIndexBlock = GetFreeBlock();
    if (dataIndexBlock == -1)
    {
        Console::puts("Out of Free Blocks! Cannot create file...\n");
        return false;
    }

    int inodeIndex = GetFreeInode();
    if (inodeIndex == -1)
    {
        Console::puts("Out of inodes! Cannot create file...\n");
        return false;
    }

    inodes[inodeIndex].fs = this;
    inodes[inodeIndex].id = _file_id;
    inodes[inodeIndex].blockNo = dataIndexBlock;
    inodes[inodeIndex].size = 0;

    disk->write(INODE_LIST_BLOCK, (unsigned char *)inodes);
    disk->write(FREE_LIST_BLOCK, free_blocks);

    // Store the indices of all the data blocks used to store a file.
    unsigned char *dataIndirectBuffer;
    dataIndirectBuffer = new unsigned char[SimpleDisk::BLOCK_SIZE];
```

Figure 10: **CreateFile** method

file_system.C: FileSystem::DeleteFile(int _file_id) : This method deletes a file and frees its associated resources.

- Locates the file's inode using `LookupFile()`.
- Frees the file's block and marks the inode as unallocated.
- Writes updated structures back to disk.



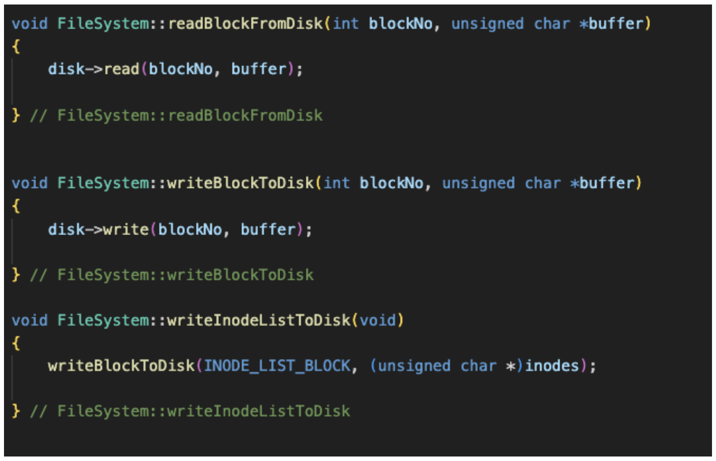
```
bool FileSystem::DeleteFile(int _file_id)
{
    Console::puts("deleting file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* First, check if the file exists. If not, throw an error.
       Then free all blocks that belong to the file and delete/invalidate
       (depending on your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

    Inode *inode = LookupFile(_file_id);
    if (inode == nullptr)
    {
        Console::puts("Failed to delete file! File does not exist.\n");
        return false;
    }

    free_blocks[inode->blockNo] = 'f';

    // Inode is free to be allocated to another block.
    inode->isAllocated = false;

    inode->blockNo = -1;
    inode->id = -1;

    writeBlockToDisk(INODE_LIST_BLOCK, (unsigned char *)inodes);
    writeBlockToDisk(FREE_LIST_BLOCK, free_blocks);

    return true;

} // FileSystem::DeleteFile
```

Figure 11: **DeleteFile** method

file_system.C: FileSystem::readBlockFromDisk(int blockNo, unsigned char *buffer) : This method reads a block of data from the disk into the provided buffer.

file_system.C: FileSystem::writeBlockToDisk(int blockNo, unsigned char *buffer) : This method writes data from the buffer into a specified disk block.

file_system.C: FileSystem::writeInodeListToDisk() : This method writes the inode list to the disk to ensure persistent storage.

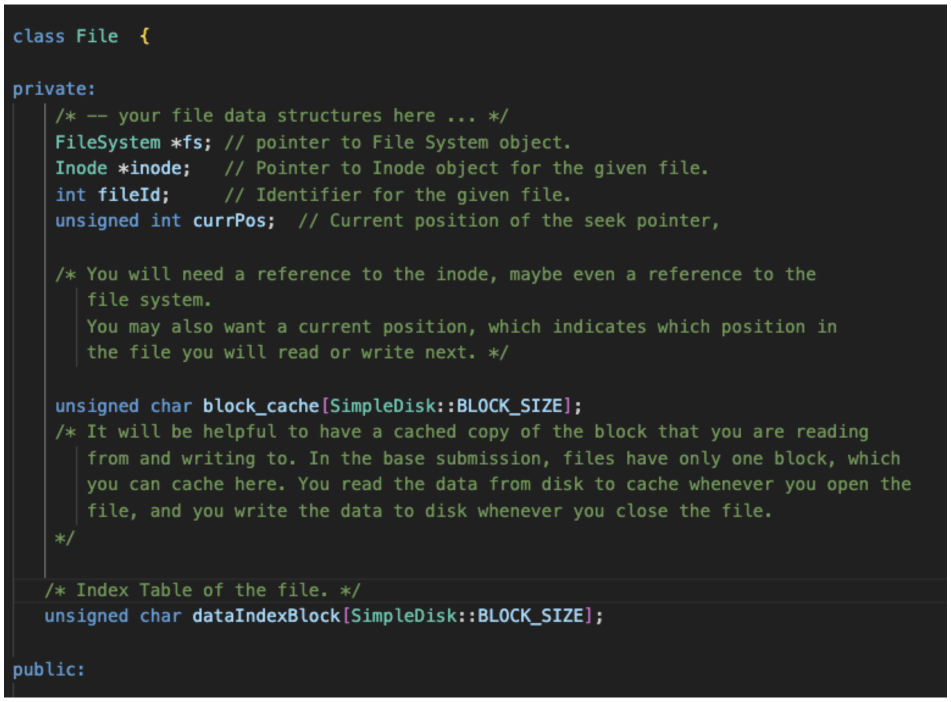


```
void FileSystem::readBlockFromDisk(int blockNo, unsigned char *buffer)
{
    disk->read(blockNo, buffer);

} // FileSystem::readBlockFromDisk


void FileSystem::writeBlockToDisk(int blockNo, unsigned char *buffer)
{
    disk->write(blockNo, buffer);

} // FileSystem::writeBlockToDisk

void FileSystem::writeInodeListToDisk(void)
{
    writeBlockToDisk(INODE_LIST_BLOCK, (unsigned char *)inodes);

} // FileSystem::writeInodeListToDisk
```

Figure 12: Read/Write block helper methods

file.H : Modifications to the header file housing class `File`:

- `FileSystem *fs`: Reference to the file system to manage file operations.
- `Inode *inode`: Reference to the inode representing the file's metadata.
- `int fileId`: Identifier for the file.
- `unsigned int currPos`: Tracks the current position in the file for read/write operations.
- `unsigned char block_cache[SimpleDisk::BLOCK_SIZE]`: Cached copy of the file's data block.
- `unsigned char dataIndexBlock[SimpleDisk::BLOCK_SIZE]`: Stores block indices for managing file data.

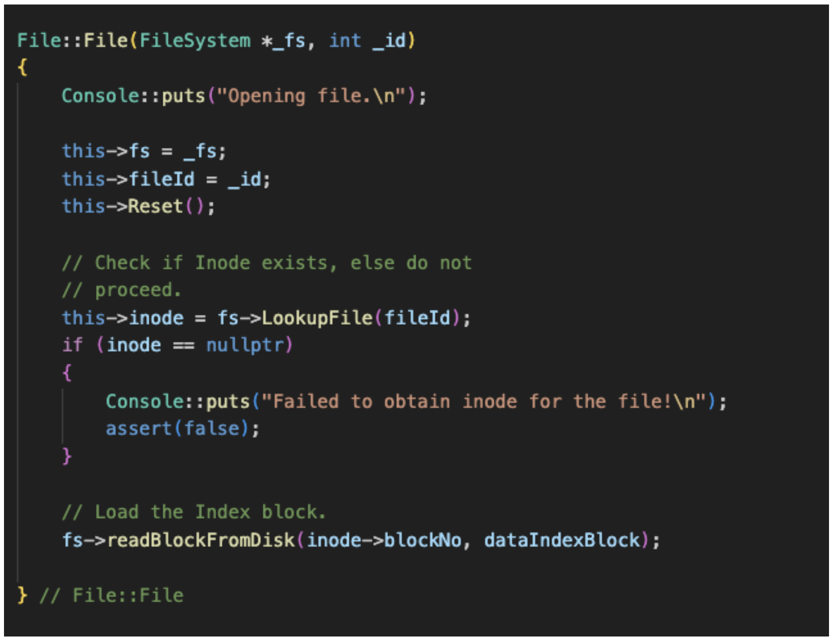


```
class File  {

private:
    /* -- your file data structures here ... */
    FileSystem *fs; // pointer to File System object.
    Inode *inode;   // Pointer to Inode object for the given file.
    int fileId;     // Identifier for the given file.
    unsigned int currPos;  // Current position of the seek pointer,

    /* You will need a reference to the inode, maybe even a reference to the
       file system.
       You may also want a current position, which indicates which position in
       the file you will read or write next. */

    unsigned char block_cache[SimpleDisk::BLOCK_SIZE];
    /* It will be helpful to have a cached copy of the block that you are reading
       from and writing to. In the base submission, files have only one block, which
       you can cache here. You read the data from disk to cache whenever you open the
       file, and you write the data to disk whenever you close the file.
    */

   /* Index Table of the file. */
   unsigned char dataIndexBlock[SimpleDisk::BLOCK_SIZE];

public:
```

Figure 13: `class File` Header

file.C: File::File(FileSystem *_fs, int _id) : This is the class constructor. It initializes a `File` instance and prepares it for operations.

- **Initialization:**
  - Assigns the `FileSystem` pointer and file ID.
  - Resets the current position to the start using the `Reset()` method.
- **Inode Check:**
  - Looks up the inode for the file using `fs->LookupFile(fileId)`.
  - If the inode is `nullptr`, the constructor asserts failure and halts execution.
- **Index Block:**
  - Loads the file's index block (managing blocks that store file data) into `dataIndexBlock` from disk using `fs->readBlockFromDisk()`.

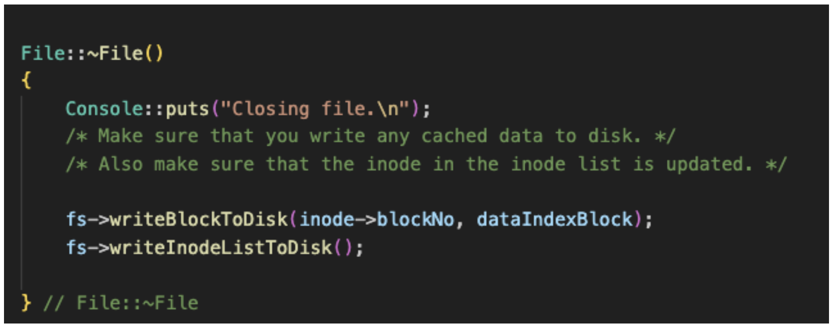


```
File::File(FileSystem *_fs, int _id)
{
    Console::puts("Opening file.\n");

    this->fs = _fs;
    this->fileId = _id;
    this->Reset();

    // Check if Inode exists, else do not
    // proceed.
    this->inode = fs->LookupFile(fileId);
    if (inode == nullptr)
    {
        Console::puts("Failed to obtain inode for the file!\n");
        assert(false);
    }

    // Load the Index block.
    fs->readBlockFromDisk(inode->blockNo, dataIndexBlock);

} // File::File
```

Figure 14: `class File` constructor

file.C: File::~File() : This the class destructor. It closes the file and ensures all changes are persisted.

- **Flush Cache:**
  - Writes the cached data index block back to disk using `fs->writeBlockToDisk()`.
- **Persist Metadata:**
  - Updates the inode list by writing it back to disk using `fs->writeInodeListToDisk()`.



```
File::~File()
{
    Console::puts("Closing file.\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */

    fs->writeBlockToDisk(inode->blockNo, dataIndexBlock);
    fs->writeInodeListToDisk();

} // File::~File
```

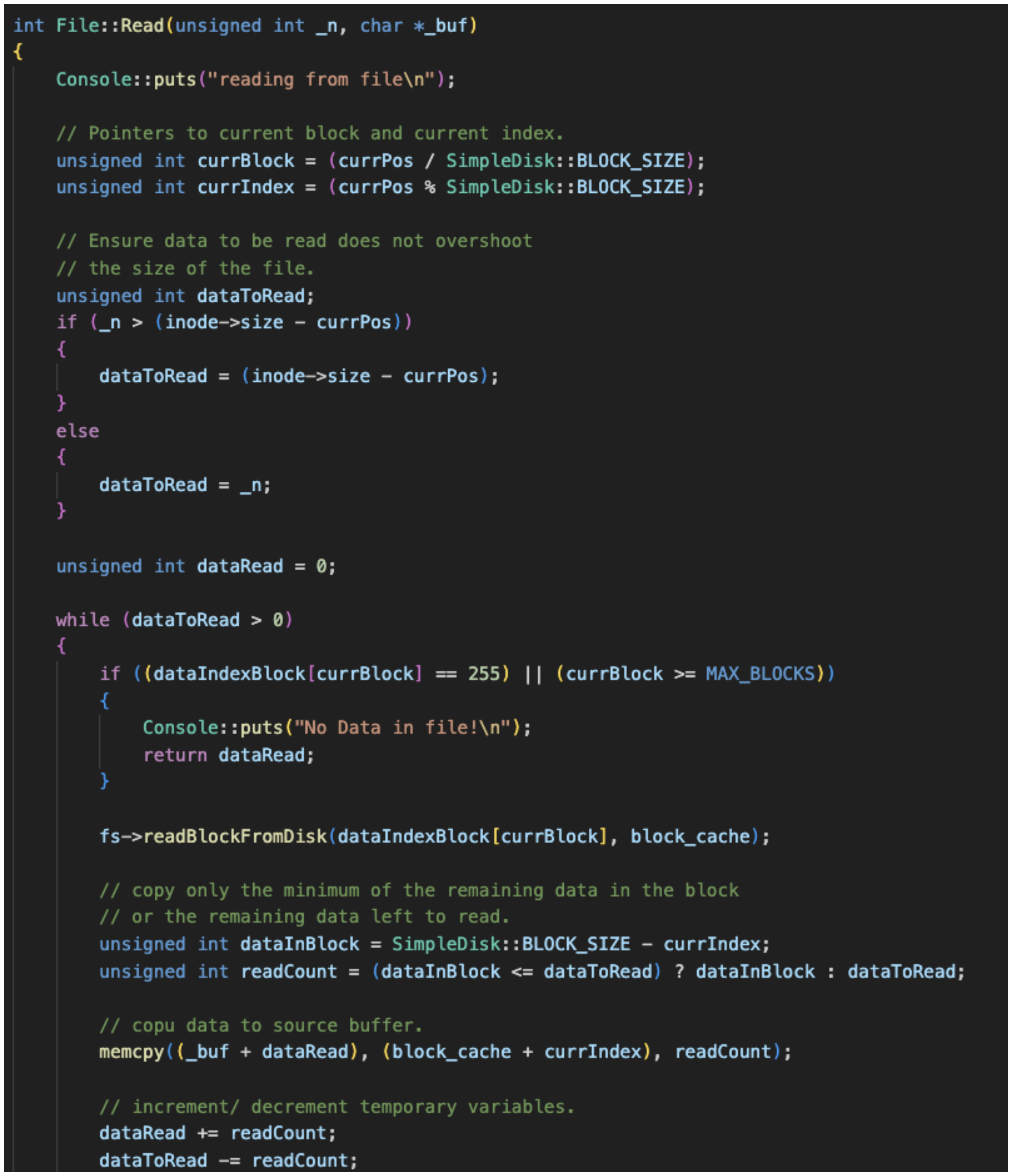
Figure 15: `class File` destructor

file.C: File::Read(unsigned int _n, char *_buf) : This method reads up to `_n` characters from the file starting at the current position and stores them in `_buf`.

- **Determine Read Size:**
  - Calculates the actual number of bytes to read based on `_n` and remaining file size (`inode->size - currPos`).
- **Read Loop:**
  - Iterates through file blocks starting at the current position.

  - For each block:
    * Reads the block from disk into `block_cache`.
    * Copies the appropriate number of bytes to `_buf`.
  - Updates `currPos`, `dataRead`, and `dataToRead` accordingly.

- **Boundary Conditions:**
  - Stops reading if the current block is not allocated or exceeds `MAX_BLOCKS`.



```
int File::Read(unsigned int _n, char *_buf)
{
    Console::puts("reading from file\n");

    // Pointers to current block and current index.
    unsigned int currBlock = (currPos / SimpleDisk::BLOCK_SIZE);
    unsigned int currIndex = (currPos % SimpleDisk::BLOCK_SIZE);

    // Ensure data to be read does not overshoot
    // the size of the file.
    unsigned int dataToRead;
    if (_n > (inode->size - currPos))
    {
        dataToRead = (inode->size - currPos);
    }
    else
    {
        dataToRead = _n;
    }

    unsigned int dataRead = 0;

    while (dataToRead > 0)
    {
        if ((dataIndexBlock[currBlock] == 255) || (currBlock >= MAX_BLOCKS))
        {
            Console::puts("No Data in file!\n");
            return dataRead;
        }

        fs->readBlockFromDisk(dataIndexBlock[currBlock], block_cache);

        // copy only the minimum of the remaining data in the block
        // or the remaining data left to read.
        unsigned int dataInBlock = SimpleDisk::BLOCK_SIZE - currIndex;
        unsigned int readCount = (dataInBlock <= dataToRead) ? dataInBlock : dataToRead;

        // copu data to source buffer.
        memcpy((_buf + dataRead), (block_cache + currIndex), readCount);

        // increment/ decrement temporary variables.
        dataRead += readCount;
        dataToRead -= readCount;
```

Figure 16: `Read` method

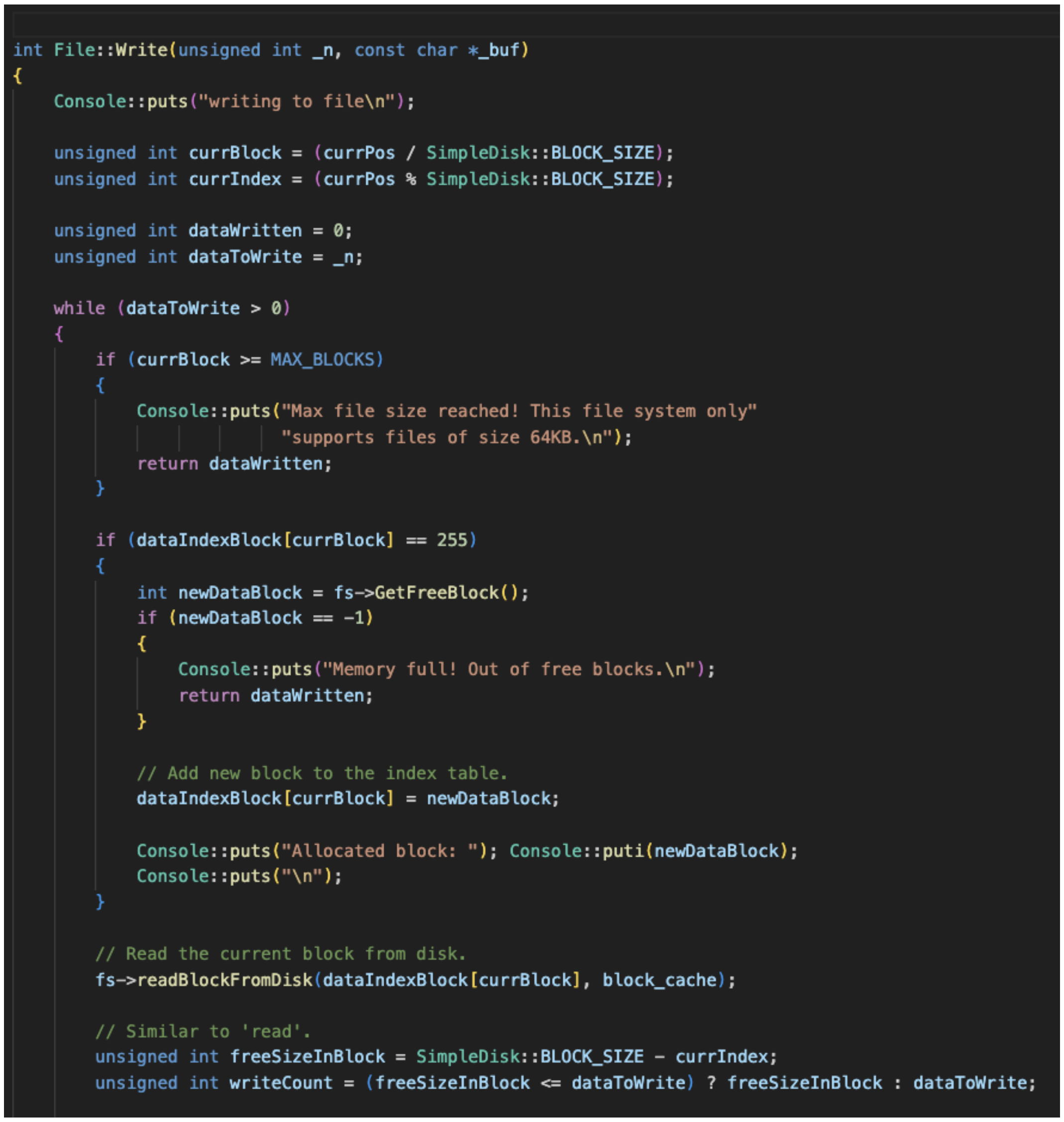
file.C: File::Write(unsigned int _n, const char *_buf) : This method writes up to `_n` characters from `_buf` to the file starting at the current position.

- **Write Loop:**

  - Iterates through file blocks starting at the current position.
  - For each block:
    * Allocates a new block if the current block is uninitialized (`dataIndexBlock[currBlock] == 255`).
    * Reads the block from disk into `block_cache`.
    * Writes data from `_buf` into `block_cache`.
    * Updates `block_cache` back to disk.

- **Update Metadata:**
  - Updates `inode->size` to reflect the new file size.

- **Boundary Conditions:**
  - Stops writing if the file size exceeds `MAX_BLOCKS` or if no free blocks are available.

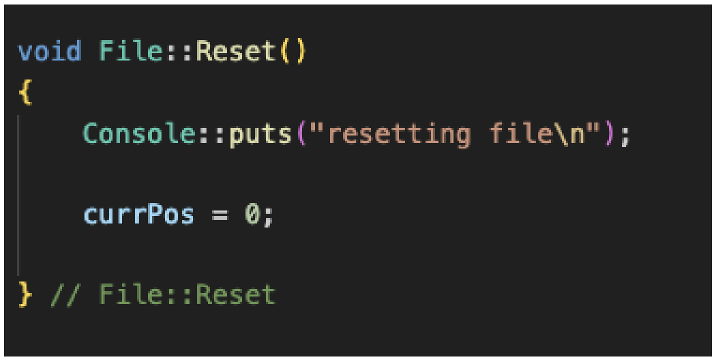


```
int File::Write(unsigned int _n, const char *_buf)
{
    Console::puts("writing to file\n");

    unsigned int currBlock = (currPos / SimpleDisk::BLOCK_SIZE);
    unsigned int currIndex = (currPos % SimpleDisk::BLOCK_SIZE);

    unsigned int dataWritten = 0;
    unsigned int dataToWrite = _n;

    while (dataToWrite > 0)
    {
        if (currBlock >= MAX_BLOCKS)
        {
            Console::puts("Max file size reached! This file system only"
                          "supports files of size 64KB.\n");
            return dataWritten;
        }

        if (dataIndexBlock[currBlock] == 255)
        {
            int newDataBlock = fs->GetFreeBlock();
            if (newDataBlock == -1)
            {
                Console::puts("Memory full! Out of free blocks.\n");
                return dataWritten;
            }

            // Add new block to the index table.
            dataIndexBlock[currBlock] = newDataBlock;

            Console::puts("Allocated block: "); Console::puti(newDataBlock);
            Console::puts("\n");
        }

        // Read the current block from disk.
        fs->readBlockFromDisk(dataIndexBlock[currBlock], block_cache);

        // Similar to 'read'.
        unsigned int freeSizeInBlock = SimpleDisk::BLOCK_SIZE - currIndex;
        unsigned int writeCount = (freeSizeInBlock <= dataToWrite) ? freeSizeInBlock : dataToWrite;
```

Figure 17: `Write` method

file.C: File::Reset() : This method resets the file's current position to the beginning (`currPos = 0`).

- Used for restarting read or write operations from the start of the file.

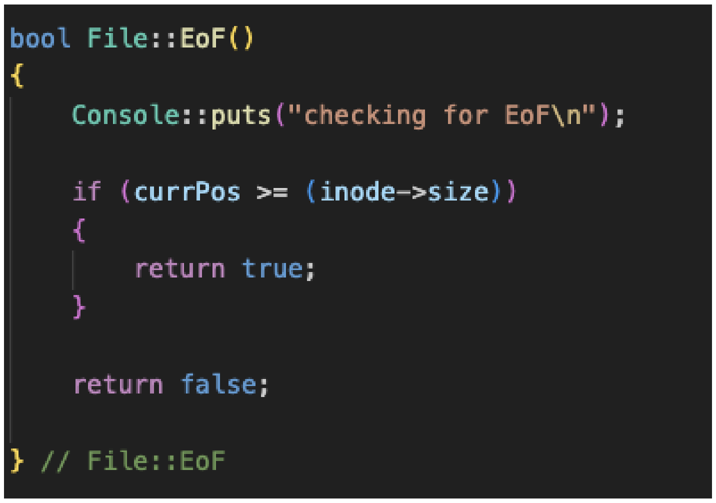


```
void File::Reset()
{
    Console::puts("resetting file\n");

    currPos = 0;

} // File::Reset
```

Figure 18: `Reset` method

bool File::EoF() : This method checks if the current position has reached or exceeded the file size (`currPos >= inode->size`).

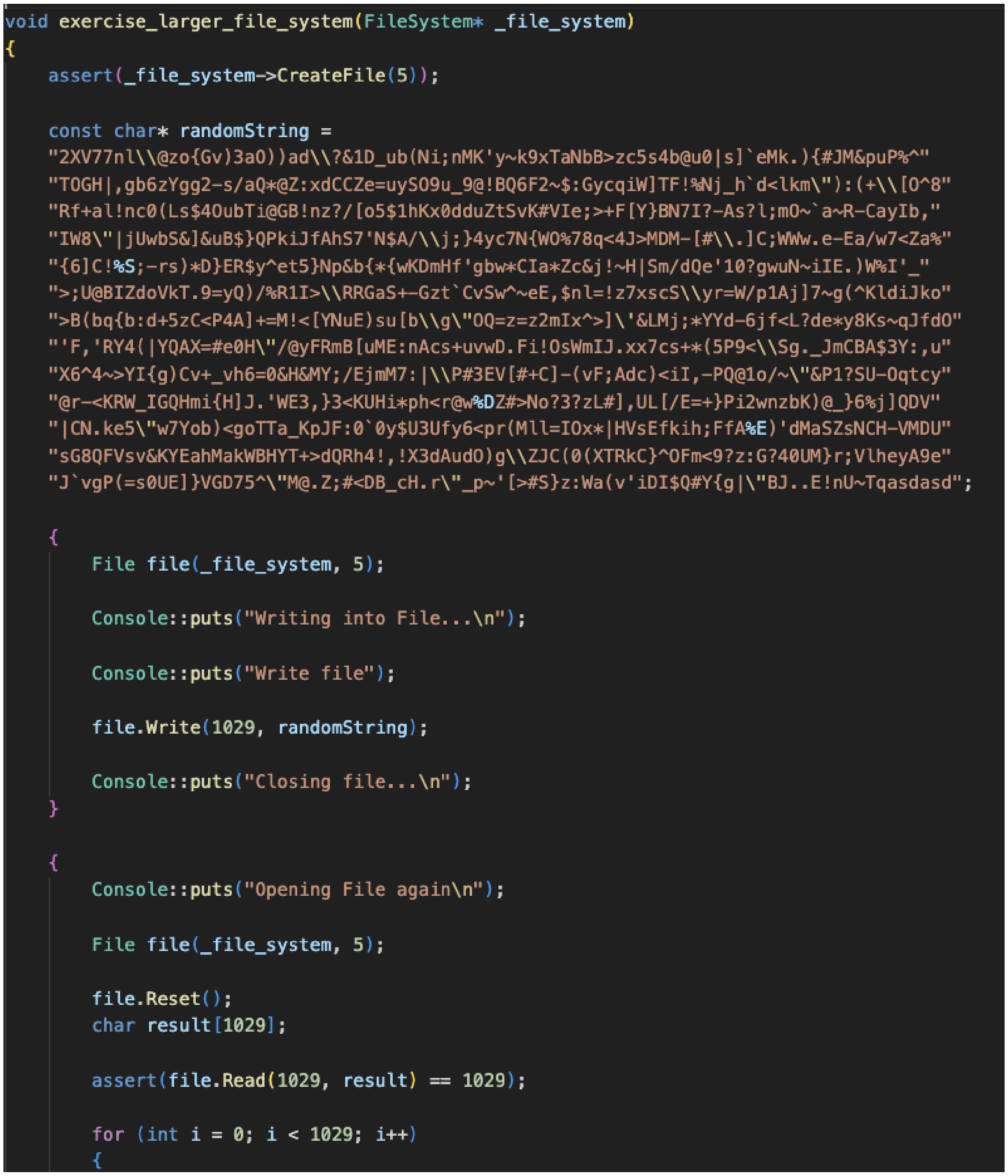
- Returns `true` if the end of the file is reached, otherwise returns `false`.



```
bool File::EoF()
{
    Console::puts("checking for EoF\n");

    if (currPos >= (inode->size))
    {
        return true;
    }

    return false;

} // File::EoF
```

Figure 19: `EoF` (End of File) method

kernel.C : The `kernel` file was modified to introduce an additional test case so as to test the functionality of storing larger files. This new method is called `exercise_larger_file_system()`

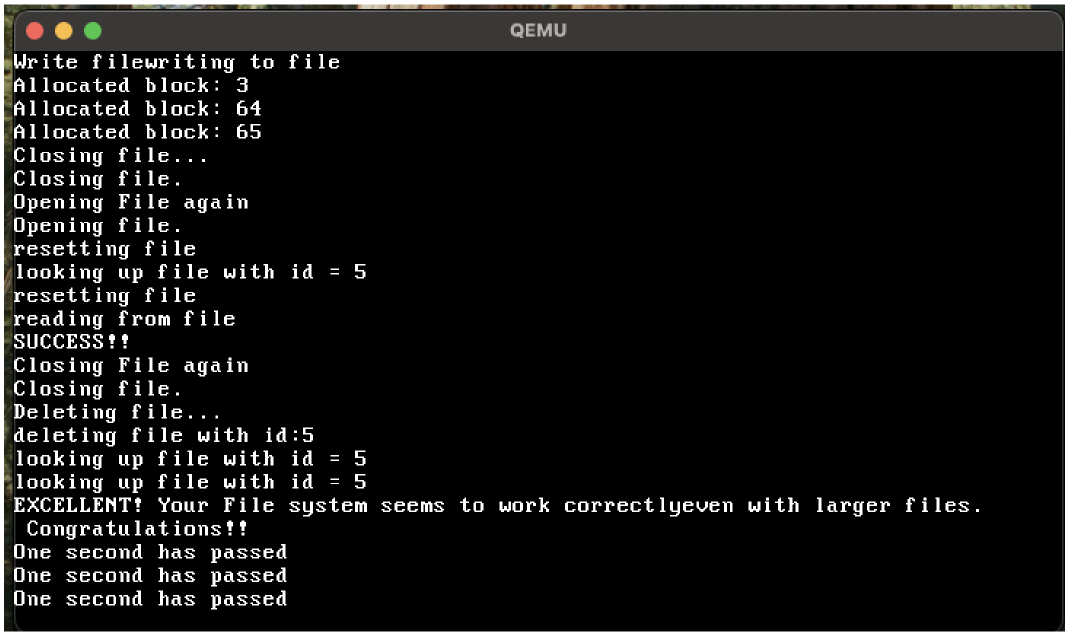


```
void exercise_larger_file_system(FileSystem* _file_system)
{
    assert(_file_system->CreateFile(5));

    const char* randomString =
    "2XV77nl\\@zo{Gv)3aO))ad\\?&1D_ub(Ni;nMK'y~k9xTaNbB>zc5s4b@u0|s]`eMk.){#JM&puP%^"
    "TOGH|,gb6zYgg2-s/aQ*@Z:xdCCZe=uySO9u_9@!BQ6F2~$:GycqiW]TF!%Nj_h`d<lkm\"):(+\\[O^8"
    "Rf+al!nc0(Ls$4OubTi@GB!nz?/[o5$1hKx0dduZtSvK#VIe;>+F[Y}BN7I?-As?l;mO~`a~R-CayIb,"
    "IW8\"|jUwbS&]&uB$}QPkiJfAhS7'N$A/\\j;}4yc7N{WO%78q<4J>MDM-[#\\.]C;WWw.e-Ea/w7<Za%"
    "{6]C!%S;-rs)*D}ER$y^et5}Np&b{*{wKDmHf'gbw*CIa*Zc&j!~H|Sm/dQe'10?gwuN~iIE.)W%I'_"
    ">;U@BIZdoVkT.9=yQ)/%R1I>\\RRGaS+-Gzt`CvSw^~eE,$nl=!z7xscS\\yr=W/p1Aj]7~g(^KldiJko"
    ">B(bq{b:d+5zC<P4A]+=M!<[YNuE)su[b\\g\"OQ=z=z2mIx^>]\'&LMj;*YYd-6jf<L?de*y8Ks~qJfdO"
    "'F,'RY4(|YQAX=#e0H\"/@yFRmB[uME:nAcs+uvwD.Fi!OsWmIJ.xx7cs+*(5P9<\\Sg._JmCBA$3Y:,u"
    "X6^4~>YI{g)Cv+_vh6=0&H&MY;/EjmM7:|\\P#3EV[#+C]-(vF;Adc)<iI,-PQ@1o/~\"&P1?SU-Oqtcy"
    "@r-<KRW_IGQHmi{H]J.'WE3,}3<KUHi*ph<r@w%DZ#>No?3?zL#],UL[/E=+}Pi2wnzbK)@_}6%j]QDV"
    "|CN.ke5\"w7Yob)<goTTa_KpJF:0`0y$U3Ufy6<pr(Mll=IOx*|HVsEfkih;FfA%E)'dMaSZsNCH-VMDU"
    "sG8QFVsv&KYEahMakWBHYT+>dQRh4!,!X3dAudO)g\\ZJC(0(XTRkC}^OFm<9?z:G?40UM}r;VlheyA9e"
    "J`vgP(=s0UE]}VGD75^\"M@.Z;#<DB_cH.r\"_p~'[>#S}z:Wa(v'iDI$Q#Y{g|\"BJ..E!nU~Tqasdasd";

    {
        File file(_file_system, 5);

        Console::puts("Writing into File...\n");

        Console::puts("Write file");

        file.Write(1029, randomString);

        Console::puts("Closing file...\n");
    }

    {
        Console::puts("Opening File again\n");

        File file(_file_system, 5);

        file.Reset();
        char result[1029];

        assert(file.Read(1029, result) == 1029);

        for (int i = 0; i < 1029; i++)
        {
```

Figure 20: `kernel` modifications

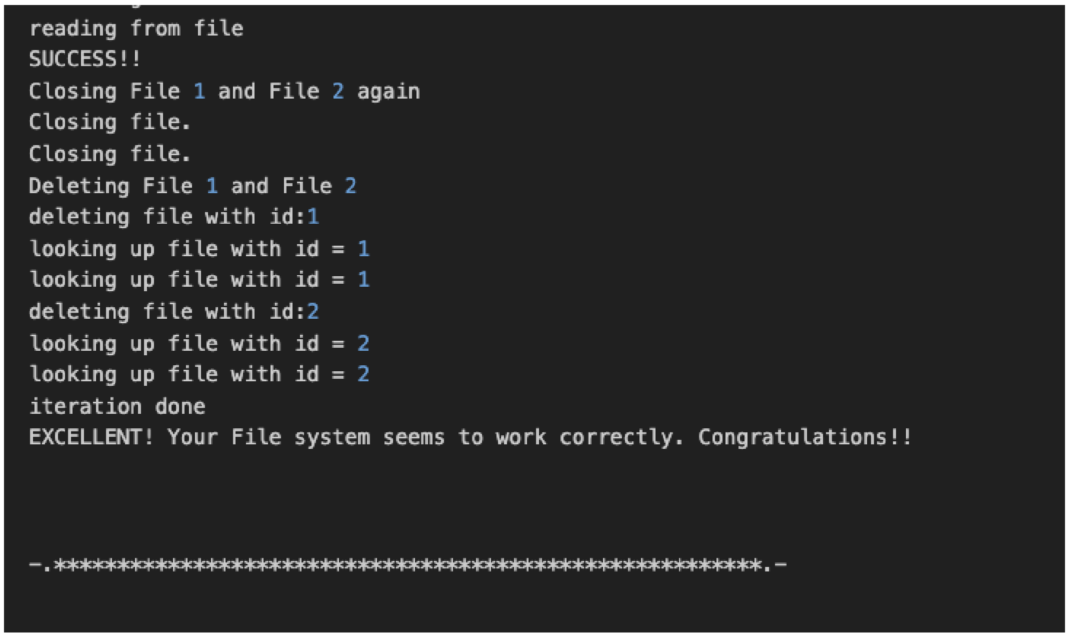
Testing

In the test-suite, `exercise_file_system()` tests the file system with smaller strings in an iterative fashion, alternatively deleting and creating files. The test method `exercise_larger_file_system` tests the file system with a larger string to show how files up to 64 KB could be stored. Detailed logs are found under *../outputs/test-run.log*



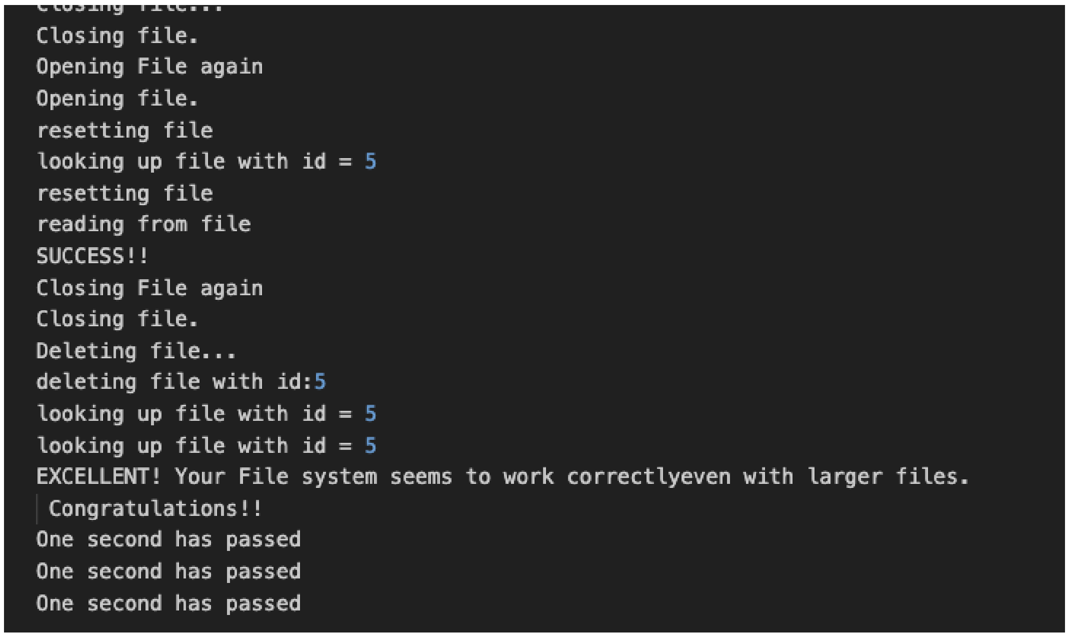
```
Write filewriting to file
Allocated block: 3
Allocated block: 64
Allocated block: 65
Closing file...
Closing file.
Opening File again
Opening file.
resetting file
looking up file with id = 5
resetting file
reading from file
SUCCESS!!
Closing File again
Closing file.
Deleting file...
deleting file with id:5
looking up file with id = 5
looking up file with id = 5
EXCELLENT! Your File system seems to work correctlyeven with larger files.
 Congratulations!!
One second has passed
One second has passed
One second has passed
```

Figure 21: QEMU output



```
reading from file
SUCCESS!!
Closing File 1 and File 2 again
Closing file.
Closing file.
Deleting File 1 and File 2
deleting file with id:1
looking up file with id = 1
looking up file with id = 1
deleting file with id:2
looking up file with id = 2
looking up file with id = 2
iteration done
EXCELLENT! Your File system seems to work correctly. Congratulations!!



-.********************************************************.-
```

Figure 22: Test run output with smaller files



```
Closing file.
Opening File again
Opening file.
resetting file
looking up file with id = 5
resetting file
reading from file
SUCCESS!!
Closing File again
Closing file.
Deleting file...
deleting file with id:5
looking up file with id = 5
looking up file with id = 5
EXCELLENT! Your File system seems to work correctlyeven with larger files.
 Congratulations!!
One second has passed
One second has passed
One second has passed
```

Figure 23: Test run output with a larger file

Limitations

The test set up does not test the full functionality of the file system to handle files as large as 64 KB, but it is possible. A new method was introduced with a file size of about 1030 bytes to showcase that files can be stored through multiple blocks. To further extend the file system, the index table itself would have to be stored under multiple data blocks, which introduces another level of indirection.