MP7: Vanilla File System

Po-Chieh Wang
UIN: 933004975
CSCE 611: Operating System

Assigned Tasks

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

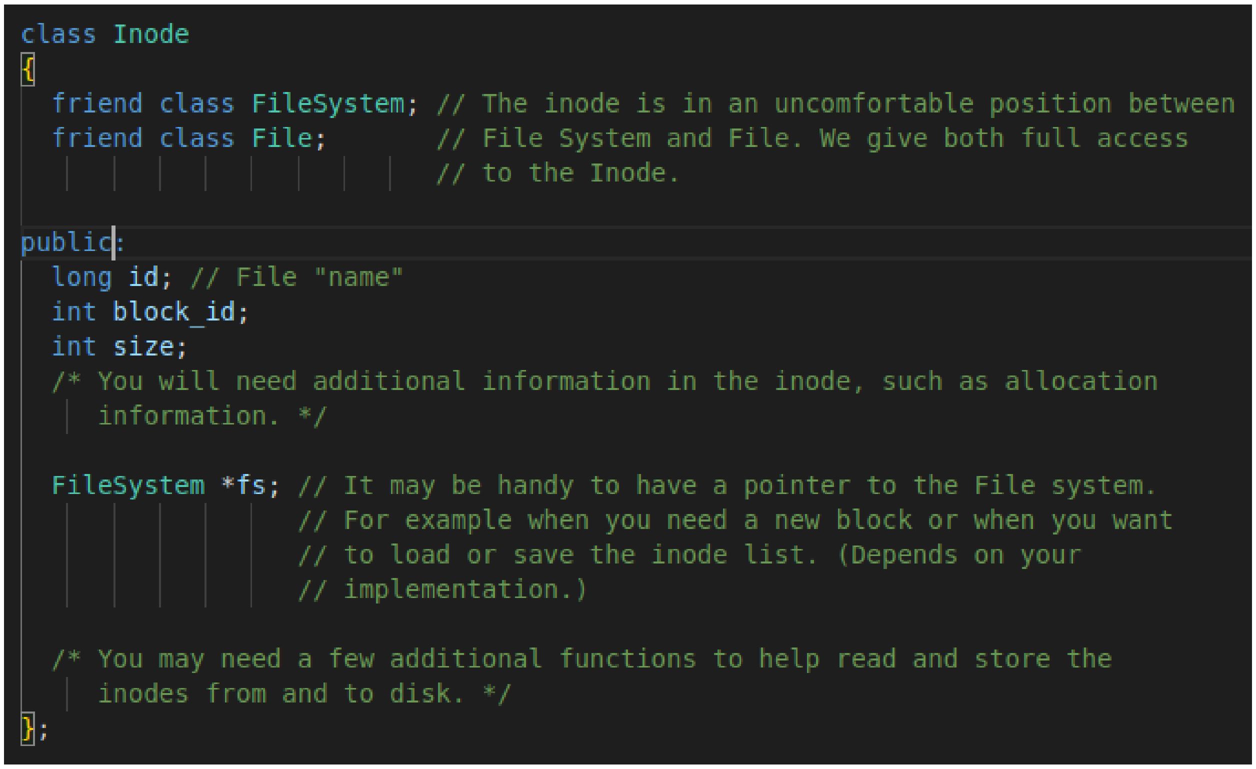
System Design

(Main and bonus option 1) The goal of this machine problem is to build a file system that can support at least 512 Byte (Main) or 64 KB(Bonus) files. To support 64 KB file size. I utilized the 0 block to store the information of the file (inodes) and the 1 block to store the information of the free blocks (bitmap). But since a 64 KB file's inode is too big that 1 block can only store 1 inode, the 0 block (inodes) only stores the information of inode other than the blocks that this file uses. Inode stores a pointer point to another block which stores the blocks that this file uses, so each file will have an inode and a block to store information.
Regarding the opening file, the file only reads or writes the first block into the block cache at first. When the file reads or writes more blocks, the file will request the next block in the disk and save this current back to the disk. In addition, there is a variable cur_pos which keeps track of the current position in case that the file reads more than size or writes more than 64 KB.

Code Description

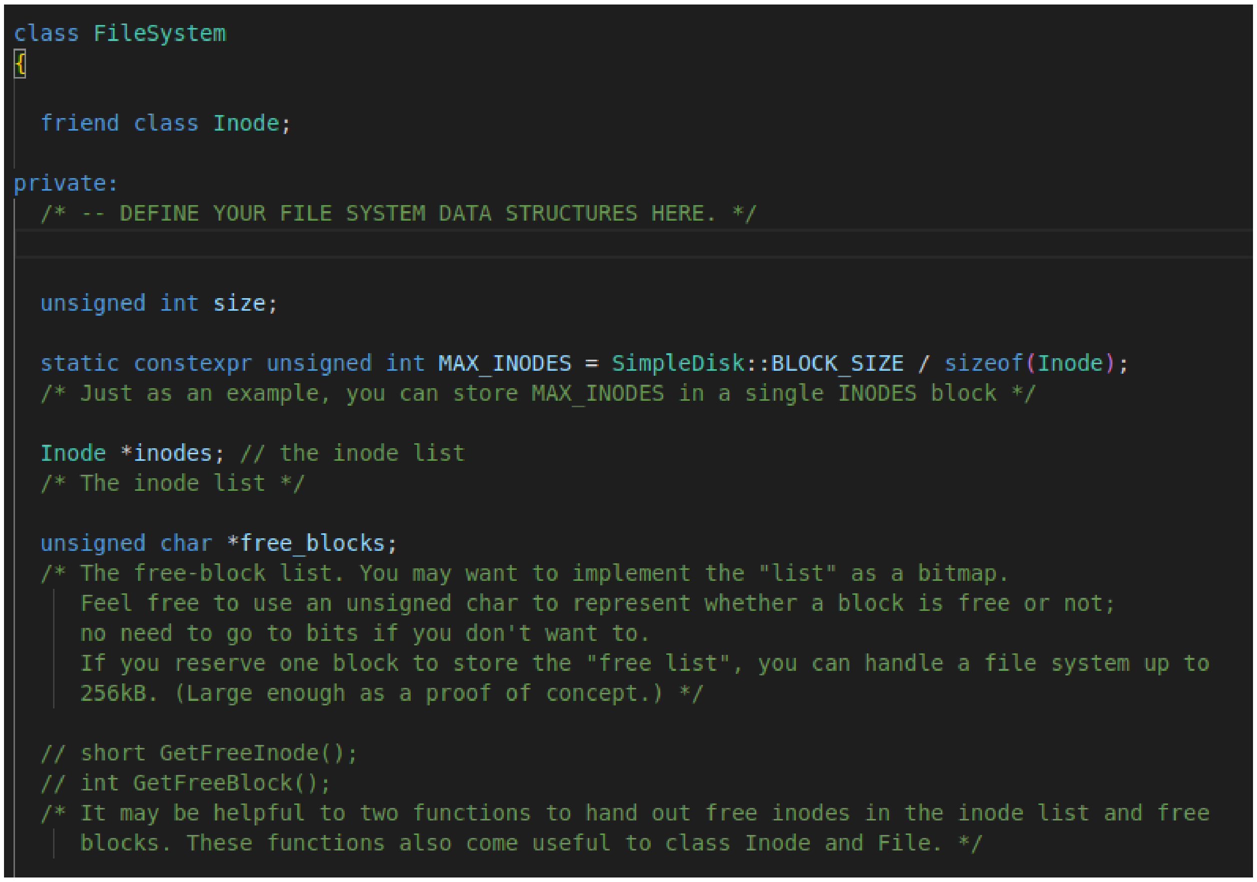
Main and bonus option 2: In this section, I implemented file_system.C, file_system.H, file.C and file.H.

file_system.H: Inode: In this class, inode is created. Inode stores the information of a file, including the size, id and related file system. Also it saves the block which stores the information of the blocks that the file is using.

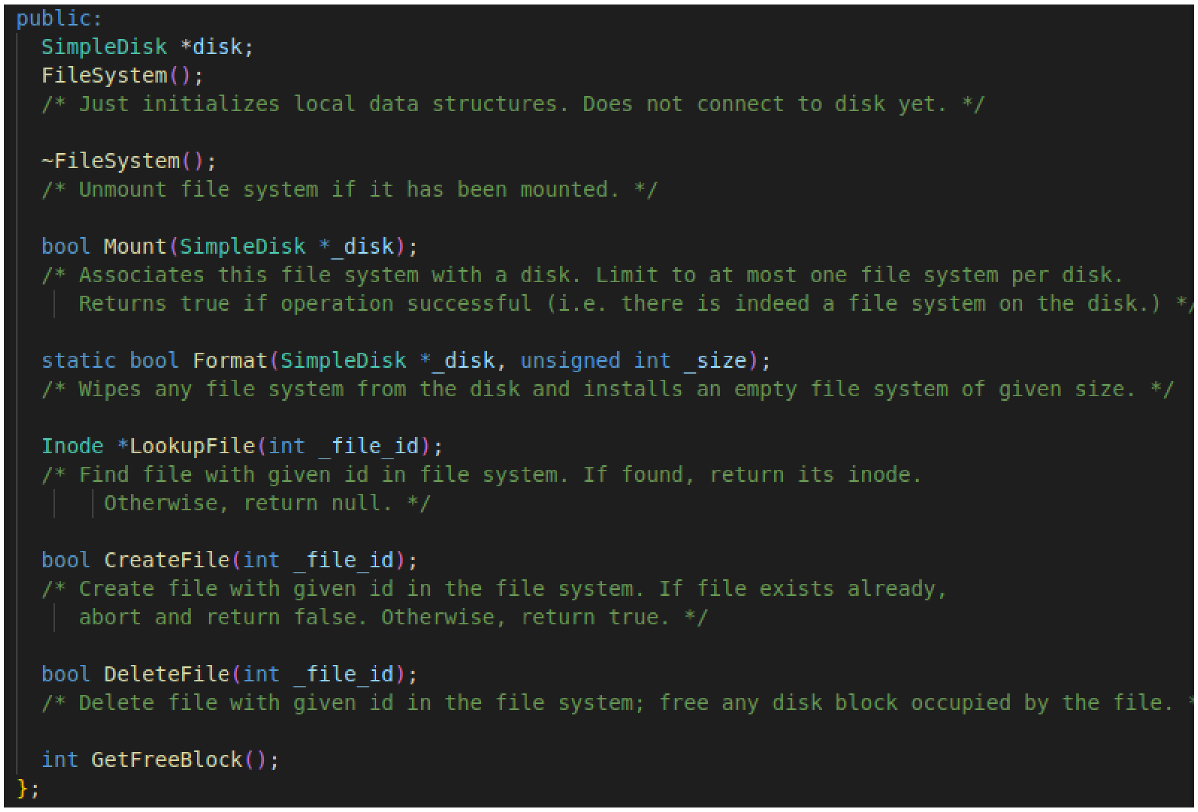


```
class Inode
{
  friend class FileSystem; // The inode is in an uncomfortable position between
  friend class File;       // File System and File. We give both full access
                           // to the Inode.

public:
  long id; // File "name"
  int block_id;
  int size;
  /* You will need additional information in the inode, such as allocation
     information. */

  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. */
};
```

file_system.H: FileSystem: This class represents the file system, it stores the information of all inodes and keeps track of the free blocks. It also connects with the disk to do some operations, such as creating file, deleting file, looking up file.

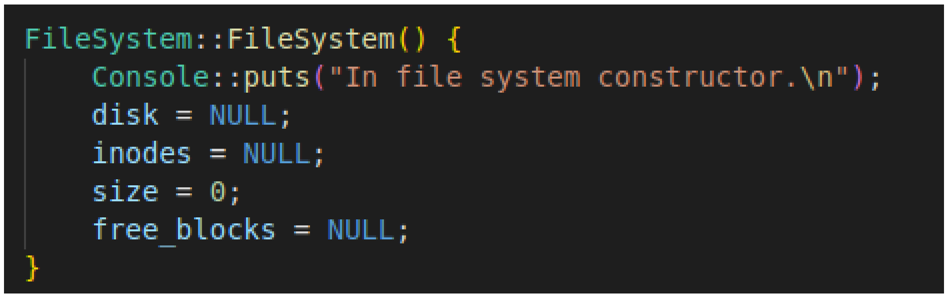


```cpp
class FileSystem
{

  friend class Inode;

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


  unsigned int size;

  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
  /* 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 as a proof of concept.) */

  // short GetFreeInode();
  // int GetFreeBlock();
  /* It may be helpful to two functions to hand out free inodes in the inode list and free
     blocks. These functions also come useful to class Inode and File. */
```



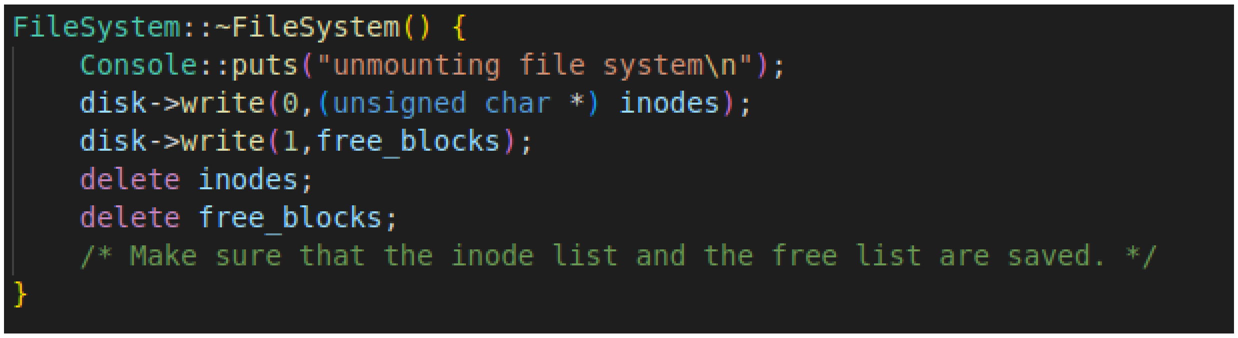
```cpp
public:
  SimpleDisk *disk;
  FileSystem();
  /* Just initializes local data structures. Does not connect to disk yet. */

  ~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.
     Returns true if operation successful (i.e. there is indeed a file system on the disk.) */

  static bool Format(SimpleDisk *_disk, unsigned int _size);
  /* Wipes any file system from the disk and installs an empty file system of given size. */

  Inode *LookupFile(int _file_id);
  /* Find file with given id in file system. If found, return its inode.
       Otherwise, return null. */

  bool CreateFile(int _file_id);
  /* Create file with given id in the file system. If file exists already,
     abort and return false. Otherwise, return true. */

  bool DeleteFile(int _file_id);
  /* Delete file with given id in the file system; free any disk block occupied by the file. *

  int GetFreeBlock();
};
```

file_system.C: FileSystem::FileSystem: This constructor basically just initializes the parameters.



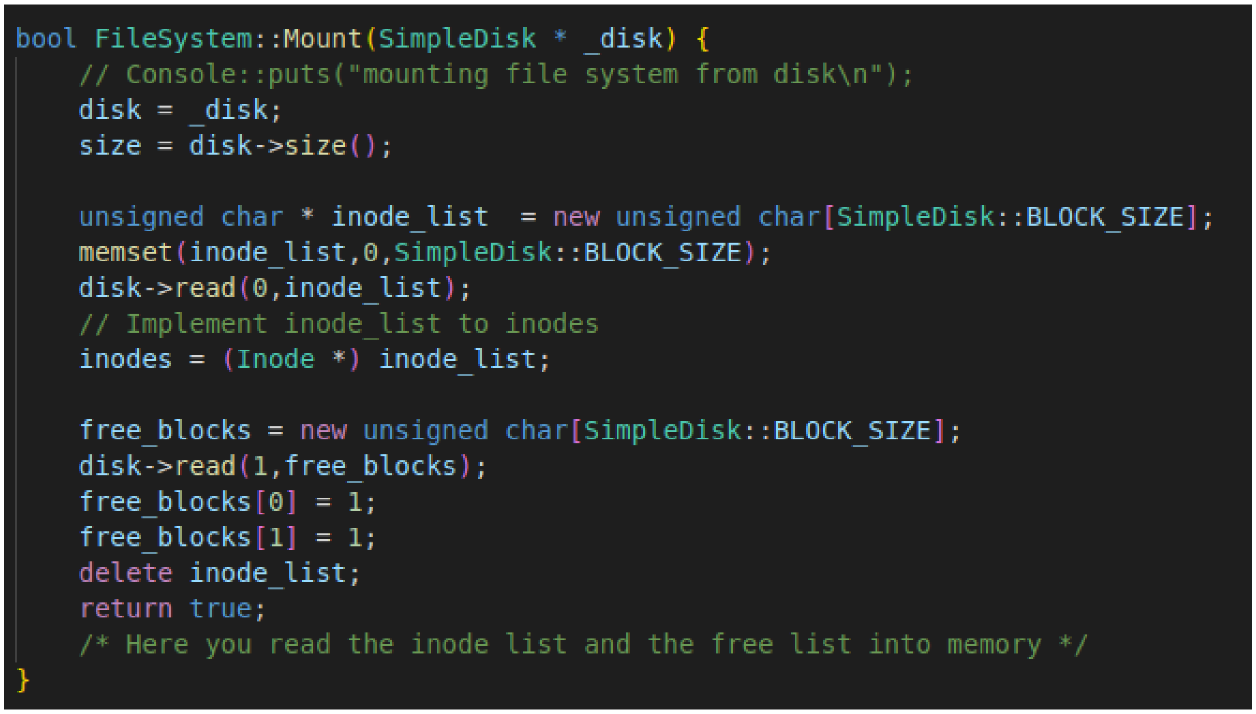
```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    disk = NULL;
    inodes = NULL;
    size = 0;
    free_blocks = NULL;
}
```

file_system.C: FileSystem::~FileSystem: This destructor saves the inodes information and the bitmap to the disk, then it deletes everything in the file system.



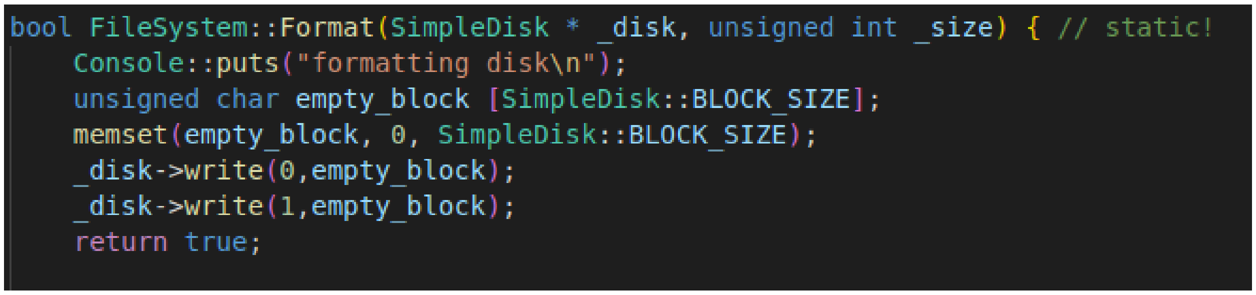
```
FileSystem::~FileSystem() {
    Console::puts("unmounting file system\n");
    disk->write(0,(unsigned char *) inodes);
    disk->write(1,free_blocks);
    delete inodes;
    delete free_blocks;
    /* Make sure that the inode list and the free list are saved. */
}
```

file_system.C: FileSystem::Mount: This function connects to the disk. Furthermore, it reads from the 0 block to get the information of inodes, 1 block to get the information of free blocks. Then it sets the 0 and 1 blocks to used.



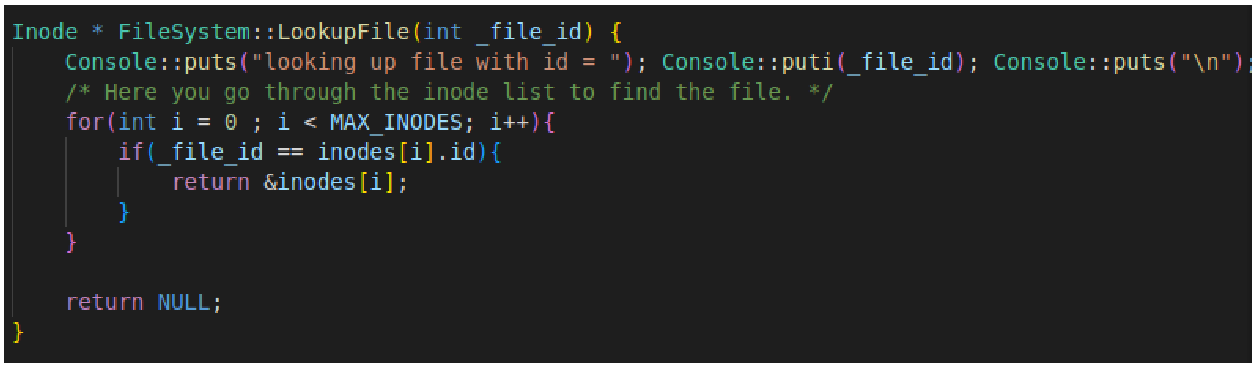
```
bool FileSystem::Mount(SimpleDisk * _disk) {
    // Console::puts("mounting file system from disk\n");
    disk = _disk;
    size = disk->size();

    unsigned char * inode_list  = new unsigned char[SimpleDisk::BLOCK_SIZE];
    memset(inode_list,0,SimpleDisk::BLOCK_SIZE);
    disk->read(0,inode_list);
    // Implement inode_list to inodes
    inodes = (Inode *) inode_list;

    free_blocks = new unsigned char[SimpleDisk::BLOCK_SIZE];
    disk->read(1,free_blocks);
    free_blocks[0] = 1;
    free_blocks[1] = 1;
    delete inode_list;
    return true;
    /* Here you read the inode list and the free list into memory */
}
```

file_system.C: FileSystem::Format: This function formats the disk by writing every byte to 0 into the 0 and 1 block in the disk.



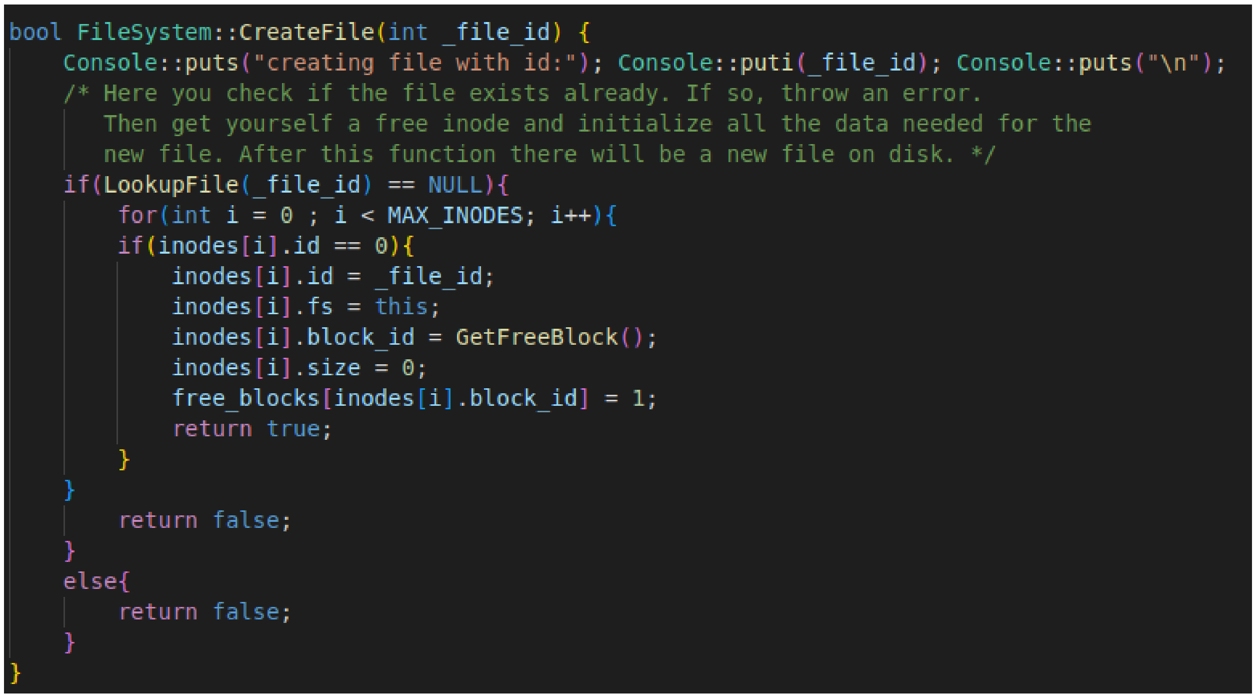
```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { // static!
    Console::puts("formatting disk\n");
    unsigned char empty_block [SimpleDisk::BLOCK_SIZE];
    memset(empty_block, 0, SimpleDisk::BLOCK_SIZE);
    _disk->write(0,empty_block);
    _disk->write(1,empty_block);
    return true;
```

file_system.C: FileSystem::LookupFile: This function looks up the inodes to check if the requested file exists or not. If it exists, it returns the inode of the file.



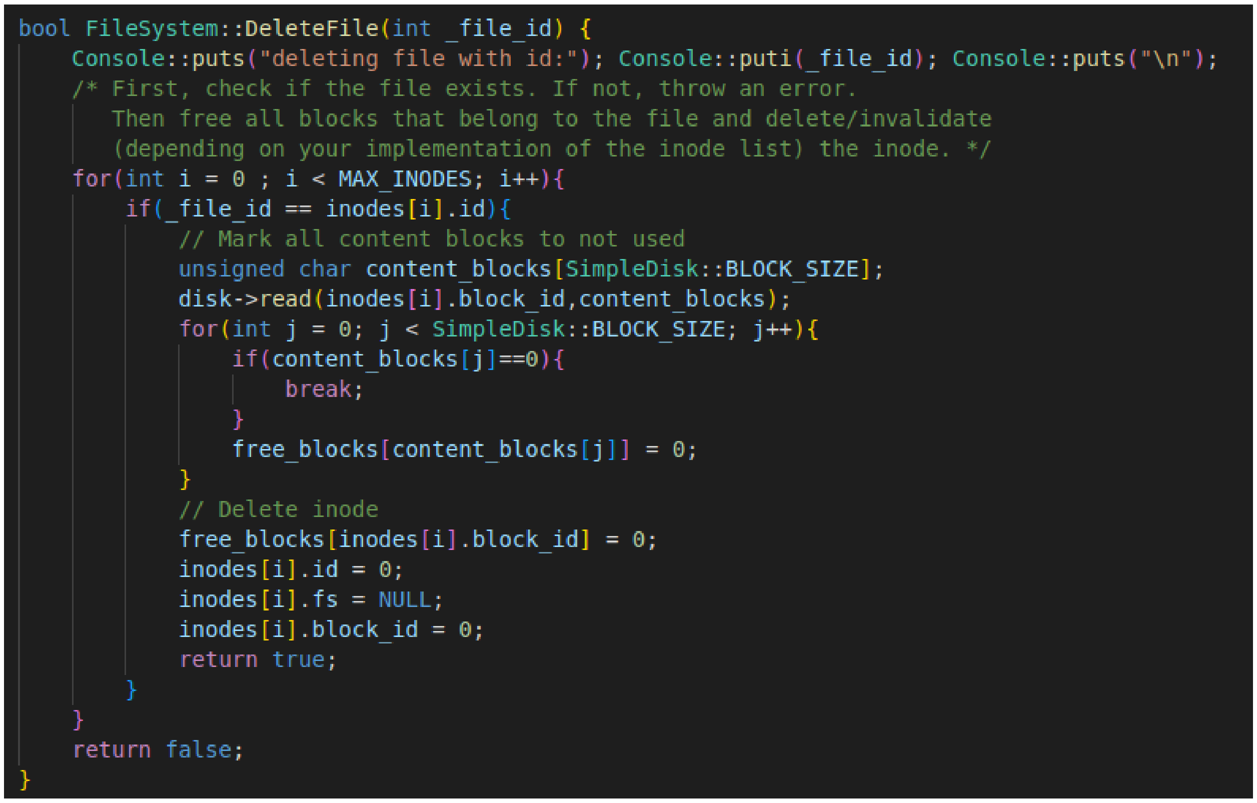
```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::puti(_file_id); Console::puts("\n")
    /* Here you go through the inode list to find the file. */
    for(int i = 0 ; i < MAX_INODES; i++){
        if(_file_id == inodes[i].id){
            return &inodes[i];
        }
    }

    return NULL;
}
```

file_system.C: FileSystem::CreateFile: This function creates the file. First, it finds the empty space to store inode. It also gets a free block to store the information of the blocks the file contains. If there is no place for creating this file, this function returns false.



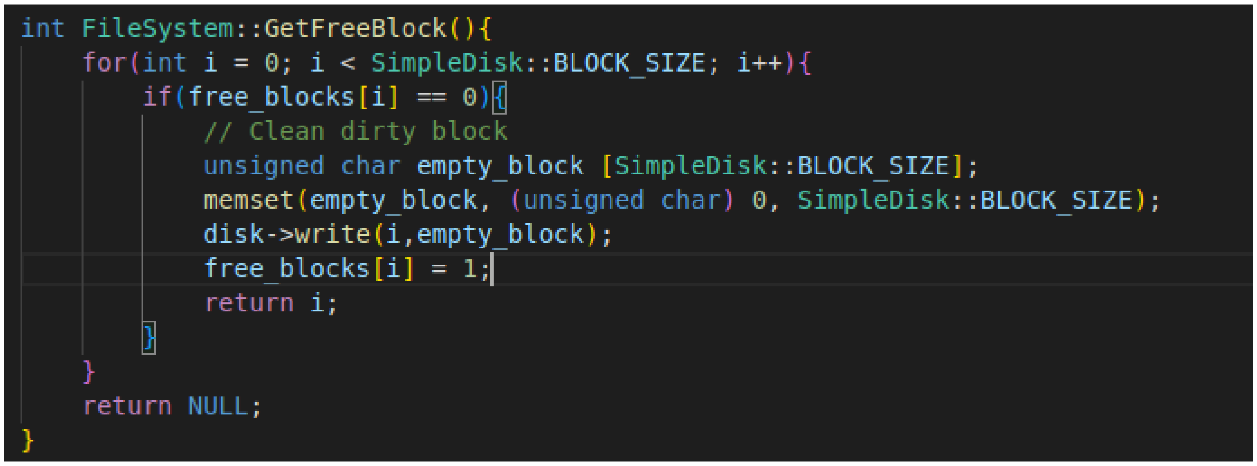
```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts("creating file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */
    if(LookupFile(_file_id) == NULL){
        for(int i = 0 ; i < MAX_INODES; i++){
        if(inodes[i].id == 0){
            inodes[i].id = _file_id;
            inodes[i].fs = this;
            inodes[i].block_id = GetFreeBlock();
            inodes[i].size = 0;
            free_blocks[inodes[i].block_id] = 1;
            return true;
        }
    }
        return false;
    }
    else{
        return false;
    }
}
```

file_system.C: FileSystem::DeleteFile: This function deletes the file. First, it checks if the file exists by using the lookup function. When the file is found, it cleans the blocks this file contains by updating the bitmap (freeblocks). Then it deletes the inode inside the inode list.



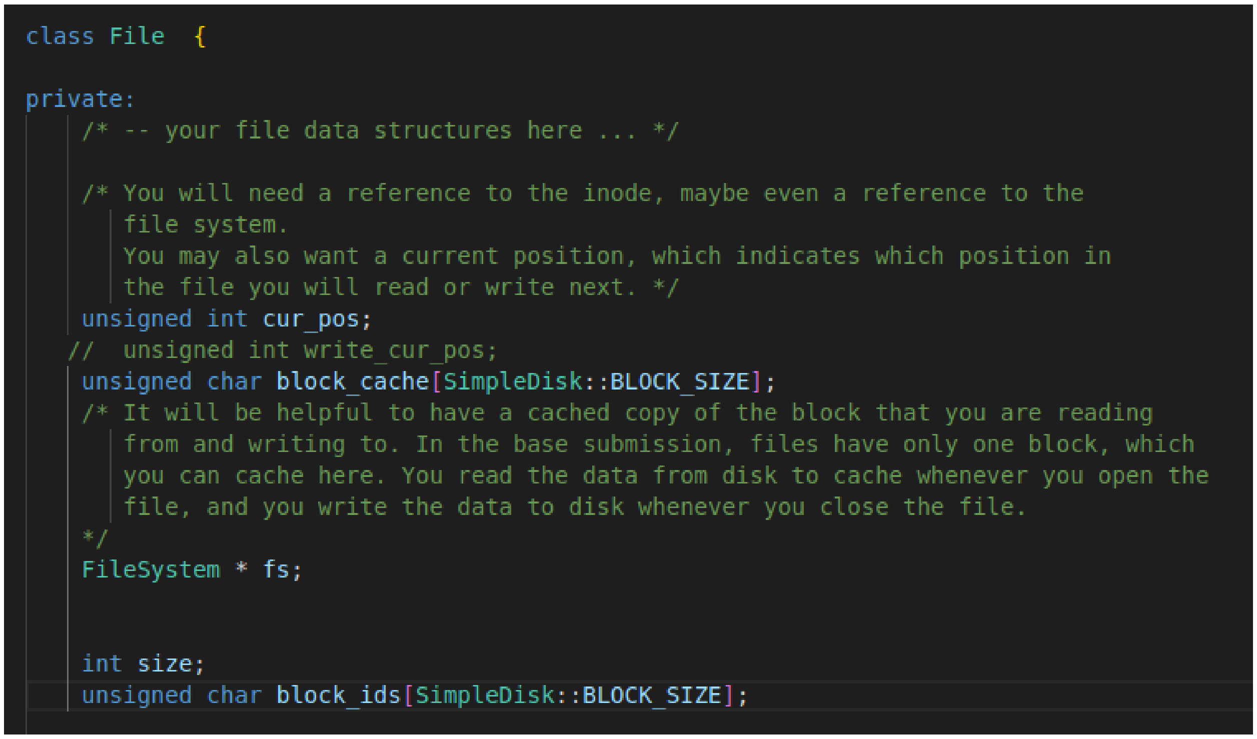
```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("deleting file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* First, check if the file exists. If not, throw an error.
       Then free all blocks that belong to the file and delete/invalidate
       (depending on your implementation of the inode list) the inode. */
    for(int i = 0 ; i < MAX_INODES; i++){
        if(_file_id == inodes[i].id){
            // Mark all content blocks to not used
            unsigned char content_blocks[SimpleDisk::BLOCK_SIZE];
            disk->read(inodes[i].block_id,content_blocks);
            for(int j = 0; j < SimpleDisk::BLOCK_SIZE; j++){
                if(content_blocks[j]==0){
                    break;
                }
                free_blocks[content_blocks[j]] = 0;
            }
            // Delete inode
            free_blocks[inodes[i].block_id] = 0;
            inodes[i].id = 0;
            inodes[i].fs = NULL;
            inodes[i].block_id = 0;
            return true;
        }
    }
    return false;
}
```

file_system.C: FileSystem::GetFreeBlock: This function searches for an unused block by looking up the freeblocks and returns the block number.

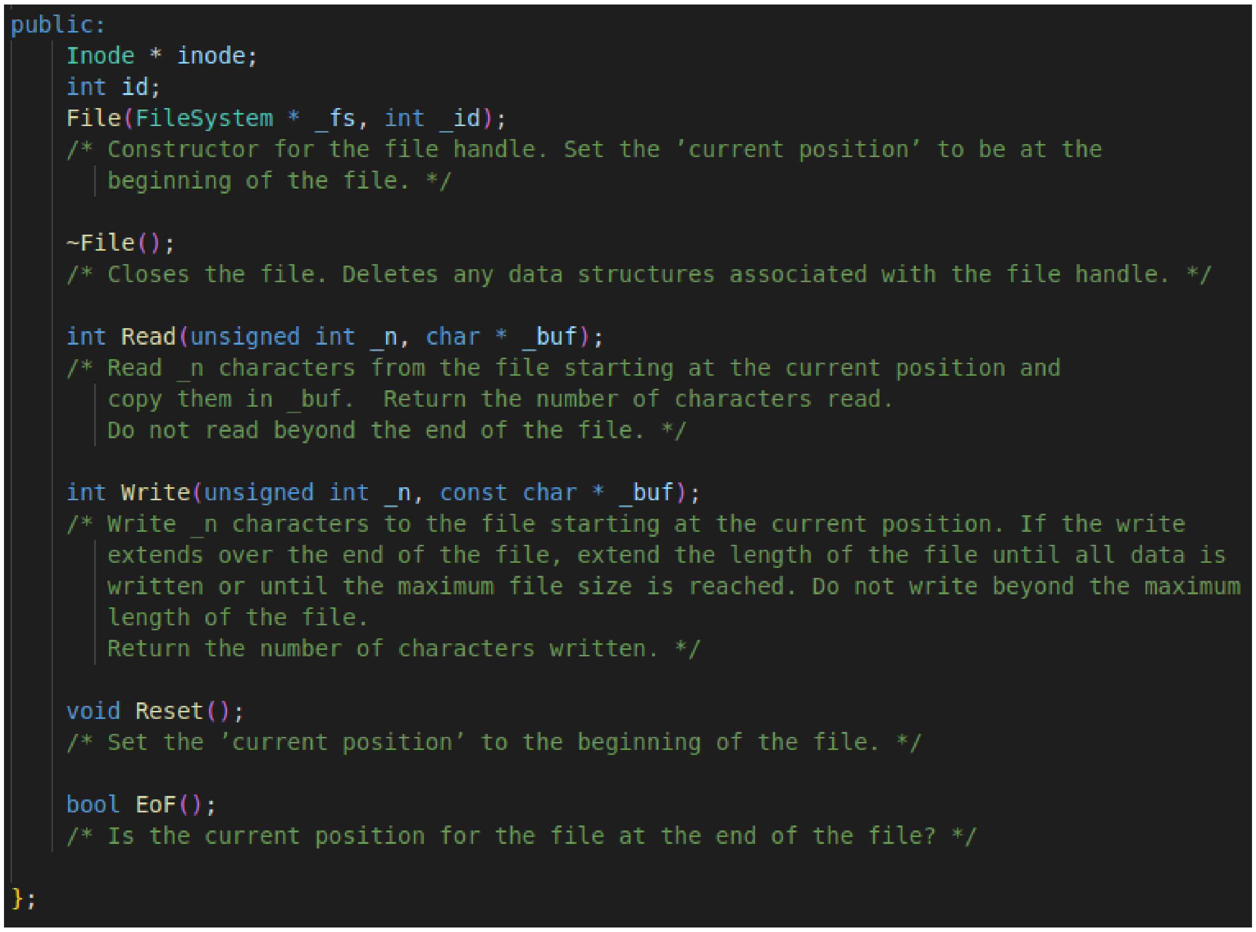


```
int FileSystem::GetFreeBlock(){
    for(int i = 0; i < SimpleDisk::BLOCK_SIZE; i++){
        if(free_blocks[i] == 0){
            // Clean dirty block
            unsigned char empty_block [SimpleDisk::BLOCK_SIZE];
            memset(empty_block, (unsigned char) 0, SimpleDisk::BLOCK_SIZE);
            disk->write(i,empty_block);
            free_blocks[i] = 1;
            return i;
        }
    }
    return NULL;
}
```

file.H: File: This class contains the information of the file and the content of the block that the current position at (block_cache). It also contains the functions to read, write, reset and check the end of the file.

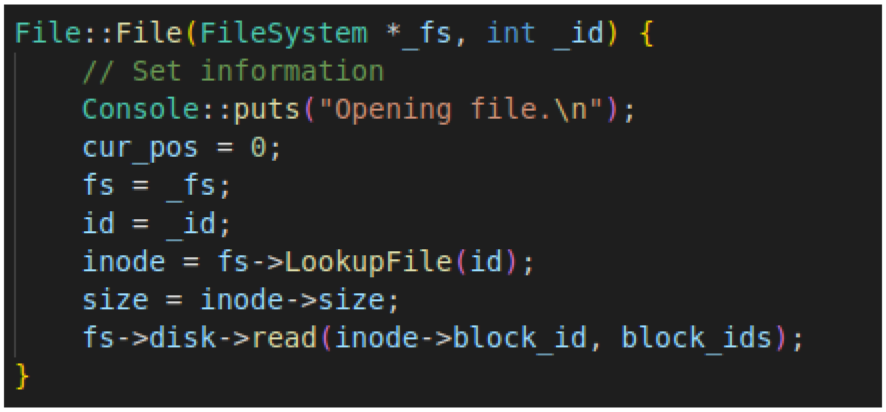


```
class File  {

private:
    /* -- your file data structures here ... */

    /* 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 int cur_pos;
   //  unsigned int write_cur_pos;
    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.
    */
    FileSystem * fs;


    int size;
    unsigned char block_ids[SimpleDisk::BLOCK_SIZE];
```



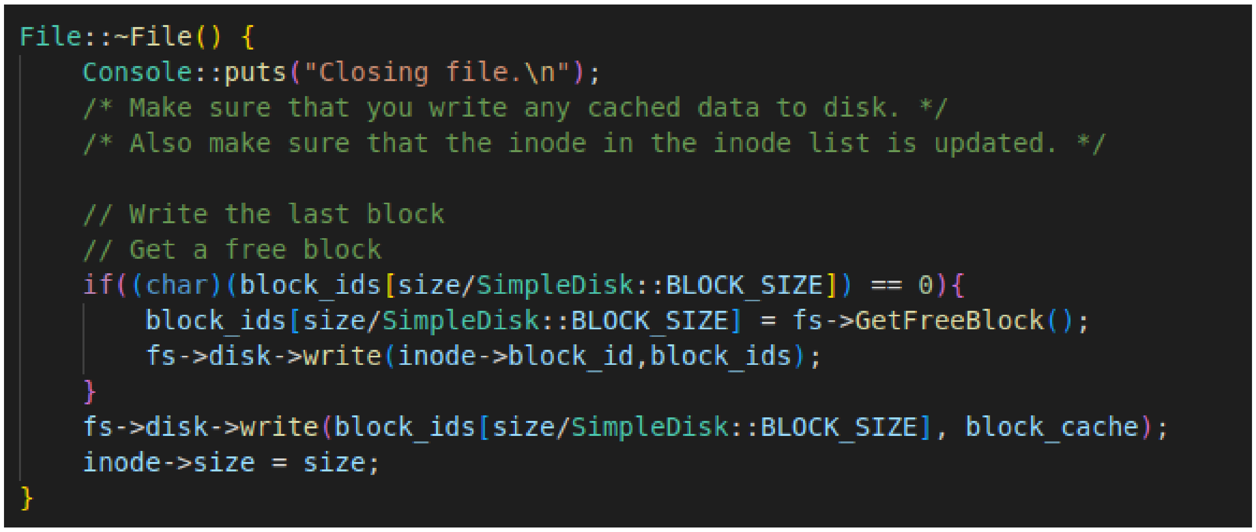
```
public:
    Inode * inode;
    int id;
    File(FileSystem * _fs, int _id);
    /* Constructor for the file handle. Set the ’current position’ to be at the
       beginning of the file. */

    ~File();
    /* Closes the file. Deletes any data structures associated with the file handle. */

    int Read(unsigned int _n, char * _buf);
    /* Read _n characters from the file starting at the current position and
       copy them in _buf.  Return the number of characters read.
       Do not read beyond the end of the file. */

    int Write(unsigned int _n, const char * _buf);
    /* Write _n characters to the file starting at the current position. If the write
       extends over the end of the file, extend the length of the file until all data is
       written or until the maximum file size is reached. Do not write beyond the maximum
       length of the file.
       Return the number of characters written. */

    void Reset();
    /* Set the ’current position’ to the beginning of the file. */

    bool EoF();
    /* Is the current position for the file at the end of the file? */

};
```

file.C: File::File: This constructor initializes the parameters, gets the inode by lookup function and gets the information of which blocks are contained by this file by reading the disk (block_ids).



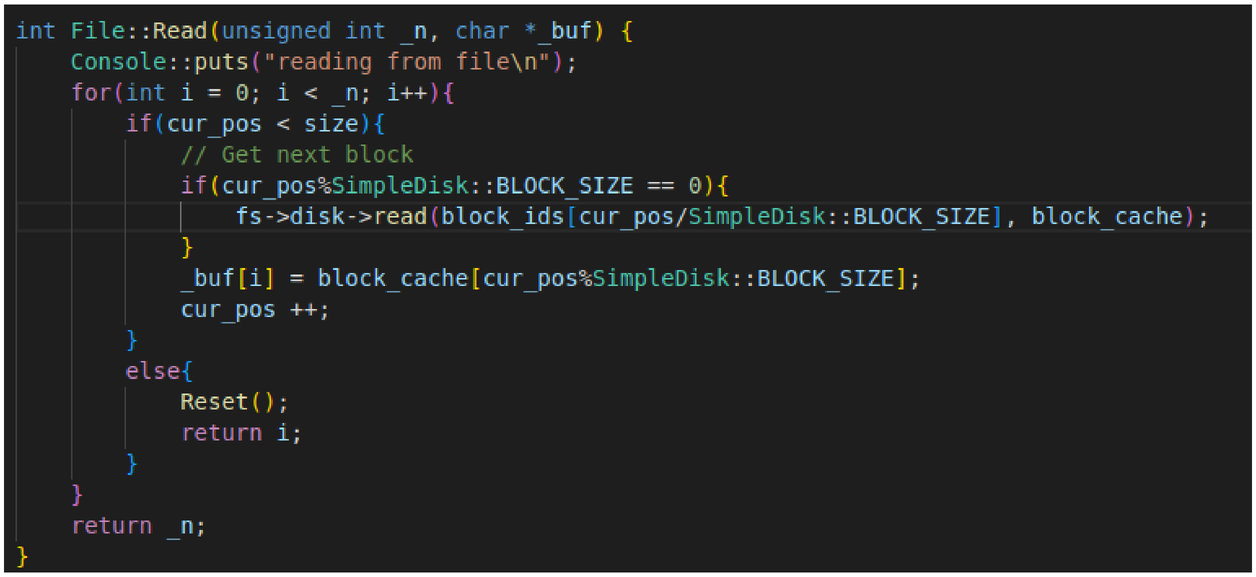
```
File::File(FileSystem *_fs, int _id) {
    // Set information
    Console::puts("Opening file.\n");
    cur_pos = 0;
    fs = _fs;
    id = _id;
    inode = fs->LookupFile(id);
    size = inode->size;
    fs->disk->read(inode->block_id, block_ids);
}
```

file.C: File::~File: This destructor writes the cache in this file to the disk. If the block is not enough, it will request 1 block by using get free block function and also update the information of containing blocks inside the disk.



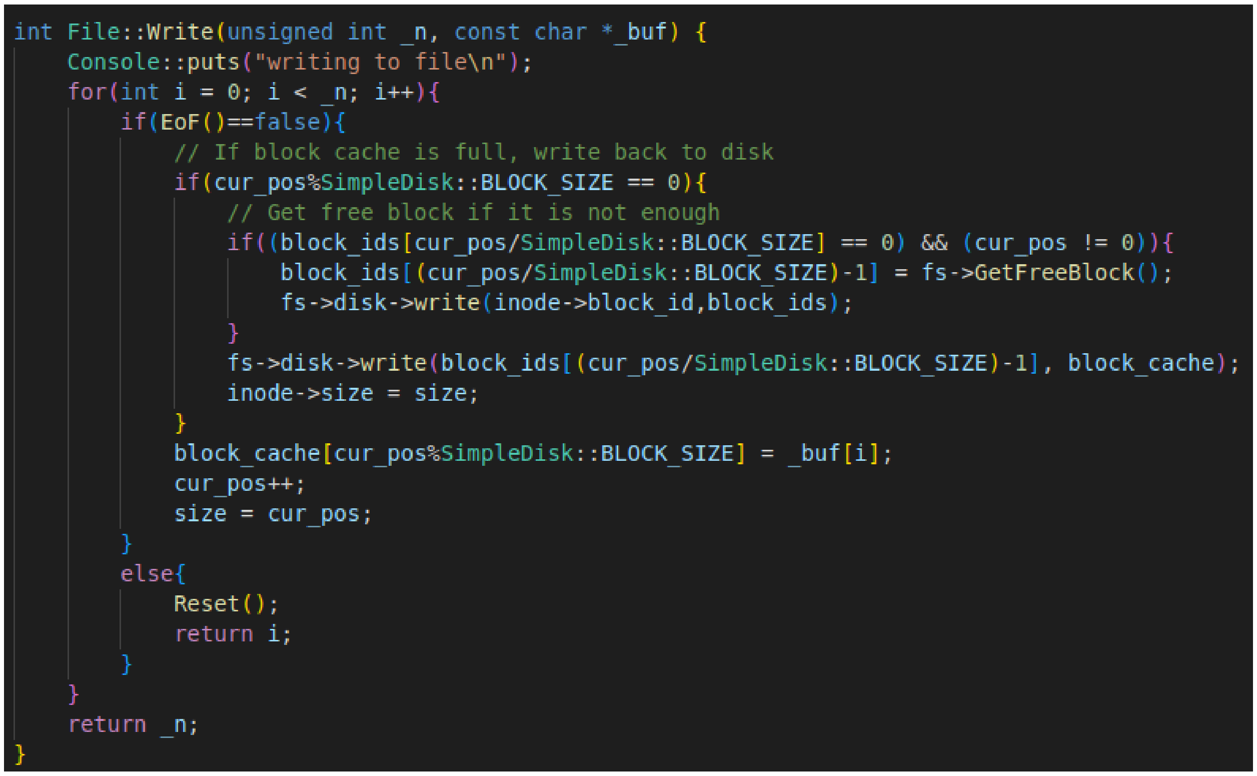
```
File::~File() {
    Console::puts("Closing file.\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */

    // Write the last block
    // Get a free block
    if((char)(block_ids[size/SimpleDisk::BLOCK_SIZE]) == 0){
        block_ids[size/SimpleDisk::BLOCK_SIZE] = fs->GetFreeBlock();
        fs->disk->write(inode->block_id,block_ids);
    }
    fs->disk->write(block_ids[size/SimpleDisk::BLOCK_SIZE], block_cache);
    inode->size = size;
}
```

file.C: File::Read: This function reads from the cache, but only can read smaller than the size of this file. If the current block_cache does not contain the information, current block_cache will read the next block from the disk.



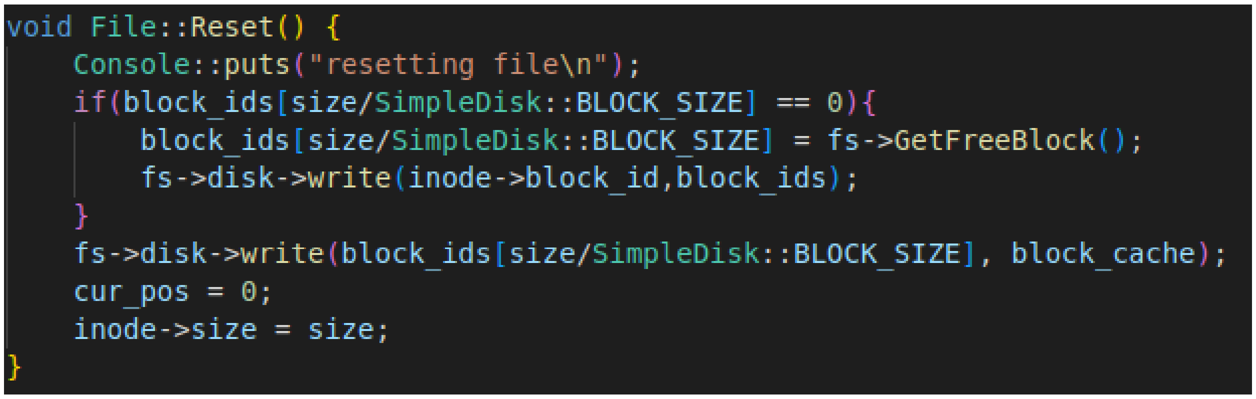
```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    for(int i = 0; i < _n; i++){
        if(cur_pos < size){
            // Get next block
            if(cur_pos%SimpleDisk::BLOCK_SIZE == 0){
                fs->disk->read(block_ids[cur_pos/SimpleDisk::BLOCK_SIZE], block_cache);
            }
            _buf[i] = block_cache[cur_pos%SimpleDisk::BLOCK_SIZE];
            cur_pos ++;
        }
        else{
            Reset();
            return i;
        }
    }
    return _n;
}
```

file.C: File::Write: This function writes the buffer to the block_cache. If it exceeds the size of block_cache, this function will first write block_cache back to the disk and request the next block. Also, it will check if it reaches the maximum size of the file which is 64 KB.



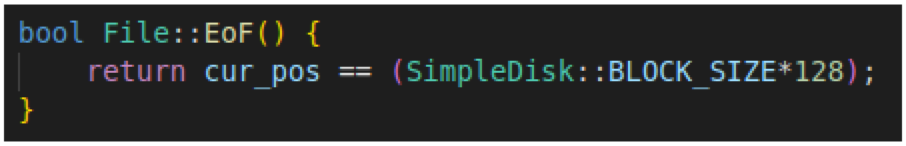
```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    for(int i = 0; i < _n; i++){
        if(EoF()==false){
            // If block cache is full, write back to disk
            if(cur_pos%SimpleDisk::BLOCK_SIZE == 0){
                // Get free block if it is not enough
                if((block_ids[cur_pos/SimpleDisk::BLOCK_SIZE] == 0) && (cur_pos != 0)){
                    block_ids[(cur_pos/SimpleDisk::BLOCK_SIZE)-1] = fs->GetFreeBlock();
                    fs->disk->write(inode->block_id,block_ids);
                }
                fs->disk->write(block_ids[(cur_pos/SimpleDisk::BLOCK_SIZE)-1], block_cache);
                inode->size = size;
            }
            block_cache[cur_pos%SimpleDisk::BLOCK_SIZE] = _buf[i];
            cur_pos++;
            size = cur_pos;
        }
        else{
            Reset();
            return i;
        }
    }
    return _n;
}
```

file.C: File::Reset: This function simply reset the current position. But it will update the current block to the disk before resetting.



```
void File::Reset() {
    Console::puts("resetting file\n");
    if(block_ids[size/SimpleDisk::BLOCK_SIZE] == 0){
        block_ids[size/SimpleDisk::BLOCK_SIZE] = fs->GetFreeBlock();
        fs->disk->write(inode->block_id,block_ids);
    }
    fs->disk->write(block_ids[size/SimpleDisk::BLOCK_SIZE], block_cache);
    cur_pos = 0;
    inode->size = size;
}
```

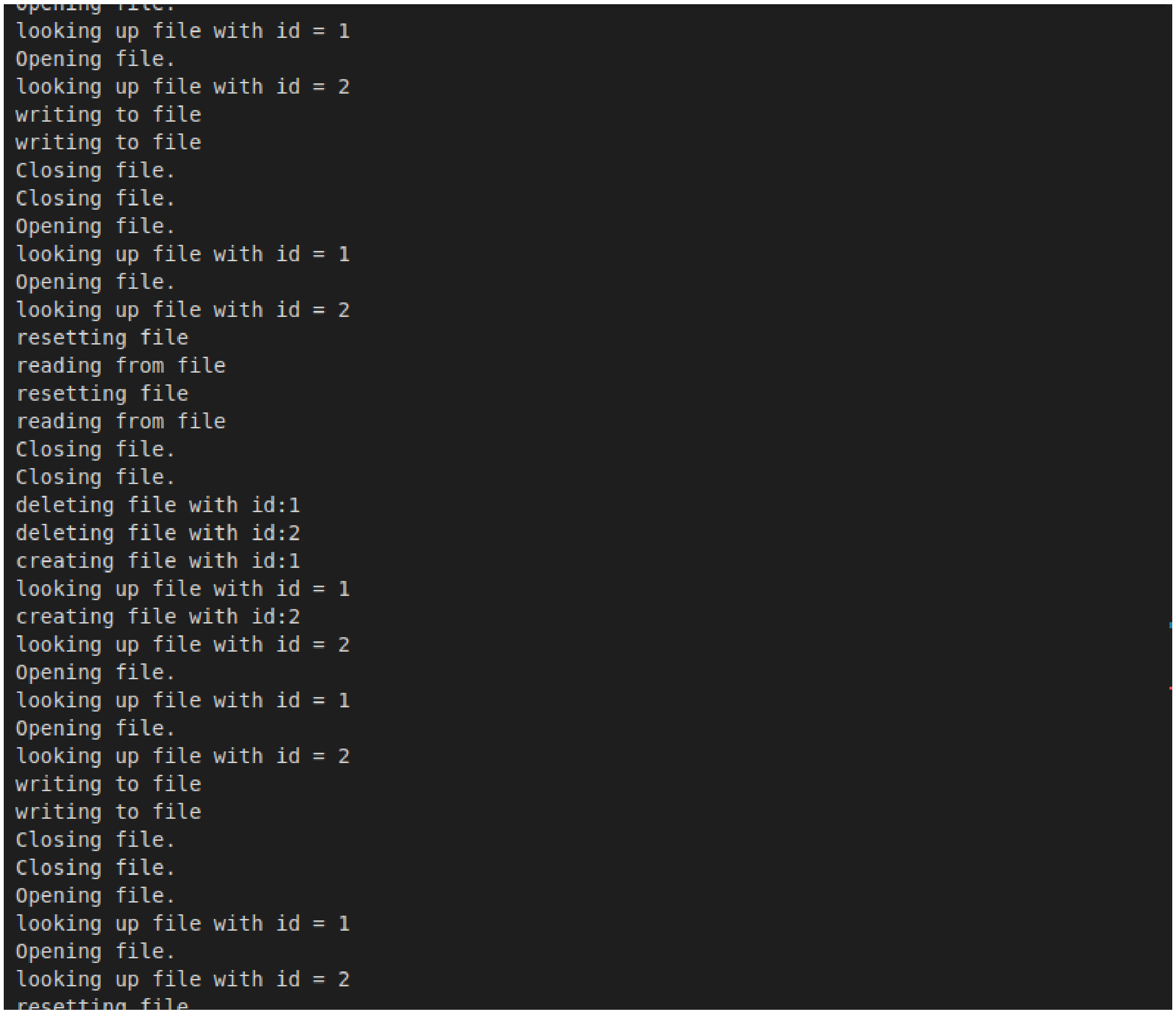
file.C: File::EoF: This function simply if the current position reaches the maximum size of the file.



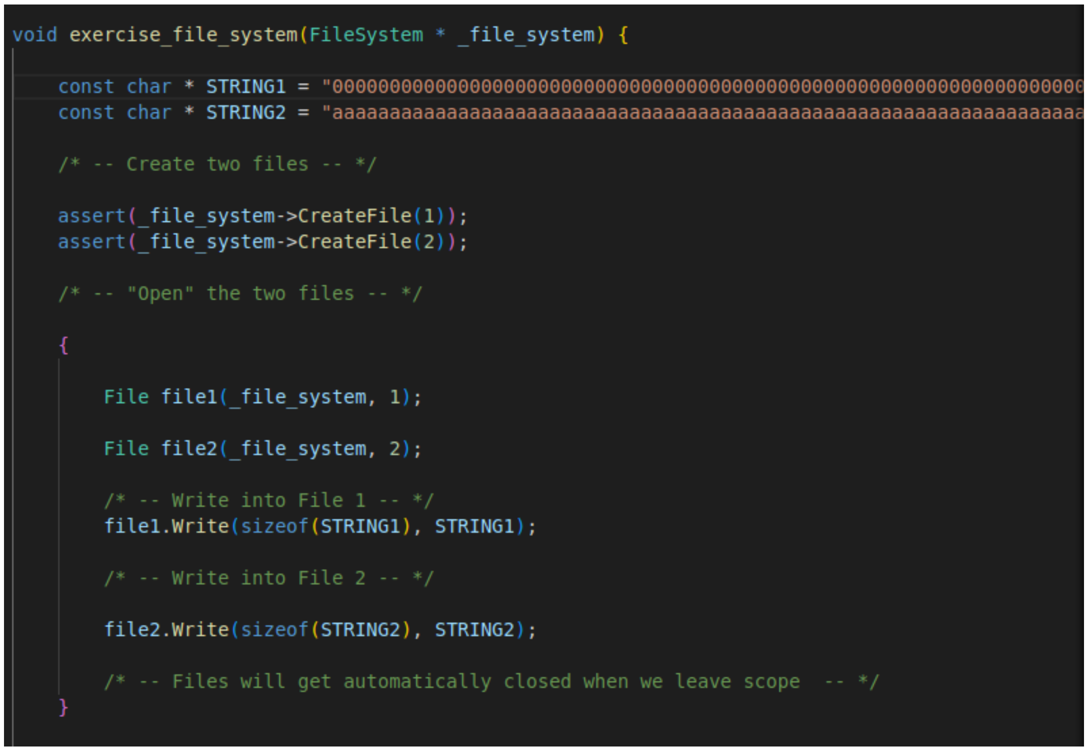
```
bool File::EoF() {
    return cur_pos == (SimpleDisk::BLOCK_SIZE*128);
}
```

Testing

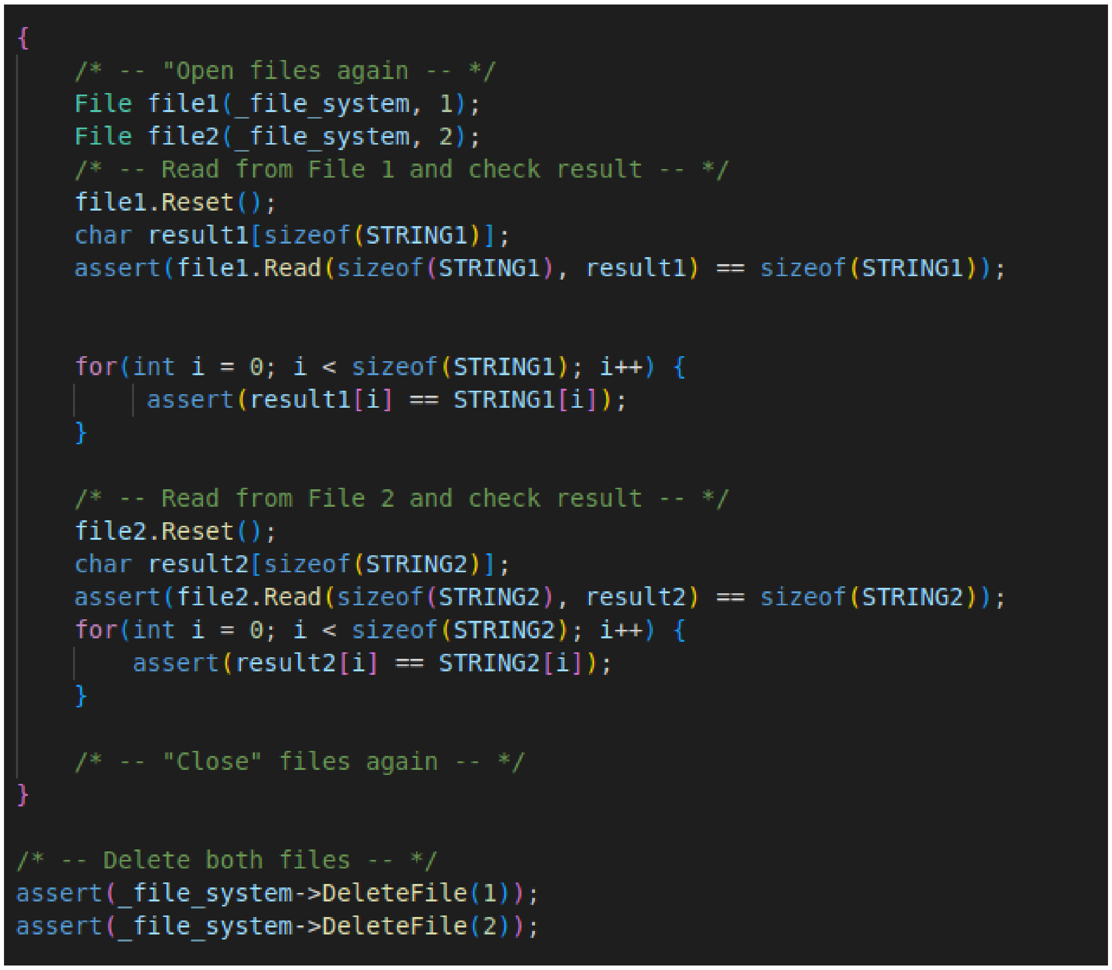
To test if my implementation can really support file with size 64 KB, I modified the kernel.C. I changed the STRING to bigger than 1 block (512 byte) and tested if this function can still run infinite times. If the function never ends, it means that the reading, writing, creating and deleting functions are correct. The result shows that my implementation is correct and can support 64 KB.



```
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file
writing to file
Closing file.
Closing file.
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
reading from file
resetting file
reading from file
Closing file.
Closing file.
deleting file with id:1
deleting file with id:2
creating file with id:1
looking up file with id = 1
creating file with id:2
looking up file with id = 2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file
writing to file
Closing file.
Closing file.
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
```



```
void exercise_file_system(FileSystem * _file_system) {

    const char * STRING1 = "00000000000000000000000000000000000000000000000000000000000000000
    const char * STRING2 = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

    /* -- Create two files -- */

    assert(_file_system->CreateFile(1));
    assert(_file_system->CreateFile(2));

    /* -- "Open" the two files -- */

    {

        File file1(_file_system, 1);

        File file2(_file_system, 2);

        /* -- Write into File 1 -- */
        file1.Write(sizeof(STRING1), STRING1);

        /* -- Write into File 2 -- */

        file2.Write(sizeof(STRING2), STRING2);

        /* -- Files will get automatically closed when we leave scope  -- */
    }
```



```
{
    /* -- "Open files again -- */
    File file1(_file_system, 1);
    File file2(_file_system, 2);
    /* -- Read from File 1 and check result -- */
    file1.Reset();
    char result1[sizeof(STRING1)];
    assert(file1.Read(sizeof(STRING1), result1) == sizeof(STRING1));


    for(int i = 0; i < sizeof(STRING1); i++) {
         assert(result1[i] == STRING1[i]);
    }

    /* -- Read from File 2 and check result -- */
    file2.Reset();
    char result2[sizeof(STRING2)];
    assert(file2.Read(sizeof(STRING2), result2) == sizeof(STRING2));
    for(int i = 0; i < sizeof(STRING2); i++) {
        assert(result2[i] == STRING2[i]);
    }

    /* -- "Close" files again -- */
}

/* -- Delete both files -- */
assert(_file_system->DeleteFile(1));
assert(_file_system->DeleteFile(2));
```