Machine Problem 7: Vanilla File System

Nachiket Umesh Naganure
UIN: 532008698
CSCE611: Operating System

Assigned Tasks

Main Task Completed.
Bonus Option 1: Completed.
Bonus Option 2: Did not attempt

System Design

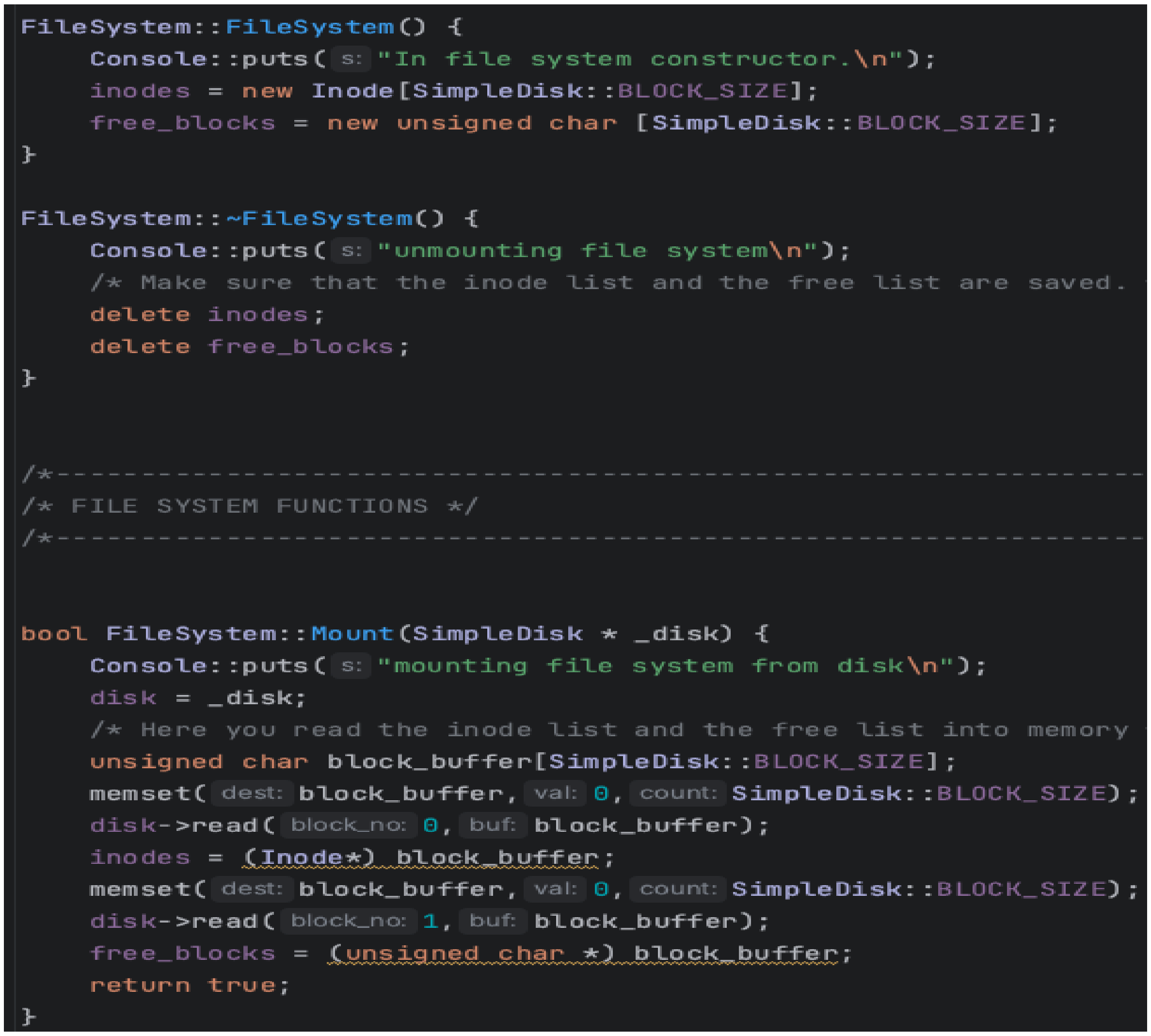
The goal of the machine problem was to design a simple vanilla file-system on top off of a simple disk.

1. **Main Task: File System and File**
    (a) File system will have to maintain a inode list and free blocks data structure
    (b) File system will have the ability to format disk
    (c) File system supports mount function
    (d) Creation of file on disk is also supported
    (e) File class will support read and write of the characters
2. **Bonus Option 1: Support for files that are up to 64kB long**
    (a) Since the files are supposed to be saved in contiguous fashion, to inode structure doesn't need changes, since it has first block and size information.
    (b) Write Function in File needs to be changed so that when its size exceeds block size, it needs to get a new block to write the extra data or characters.
    (c) The condition is (position + _n) ¿ BLOCK_SIZE, then read needs a new block.
    (d) In order for the to have contiguous blocks, the new block also needs to be contiguous.
    (e) If we find the a free block which is contiguous then all we need to do is do is mark the block as used in free_blocks array and update the inode size.
    (f) But on the flip side, if we don't find it then the following steps need to be done.
        i. Find a new set of contiguous free blocks which satisfy the requirement ie num of existing blocks + 1.
        ii. Copy the data from previous blocks to new set of contiguous blocks. Mark them as used in free_blocks array.
        iii. Release the previous blocks by marking them as unused in free_blocks array.
        iv. Update the INODE by updating its first block and size information.
        v. Continue writing the characters from buffer in file.
        vi. Also make sure if the size is exceeding 64KB return from write function.
    (g) If we are not able to find a set of free blocks which are number of previous blocks + 1, then we don't have space on disk return from write function.
    (h) The File read function also needs to modified to read beyond one block, but since the blocks are continuous, we just need to read from the next block till the last block(block ¡ inode.block + size/BLOCK_Size).

Code Description

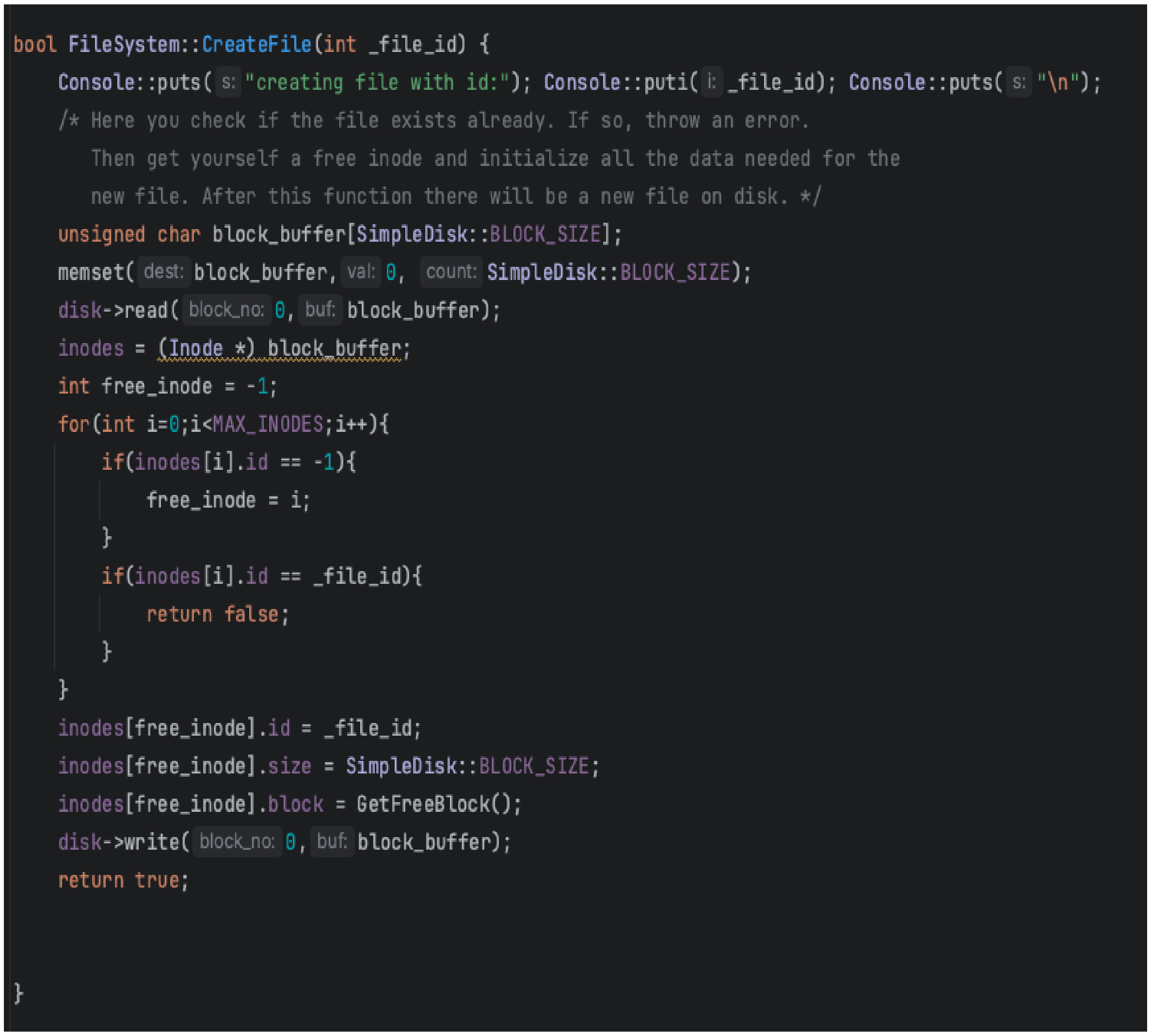
I mainly made changes in "FileSystem.c", "FileSystem.h" and "File.h", "File.c". To compile the code, you need to run the make file. To run the simulator, use copykernel.sh script and run the command "bochs -f bochsrc.bxrc" to start the bochs simulator.

FileSystem Class :



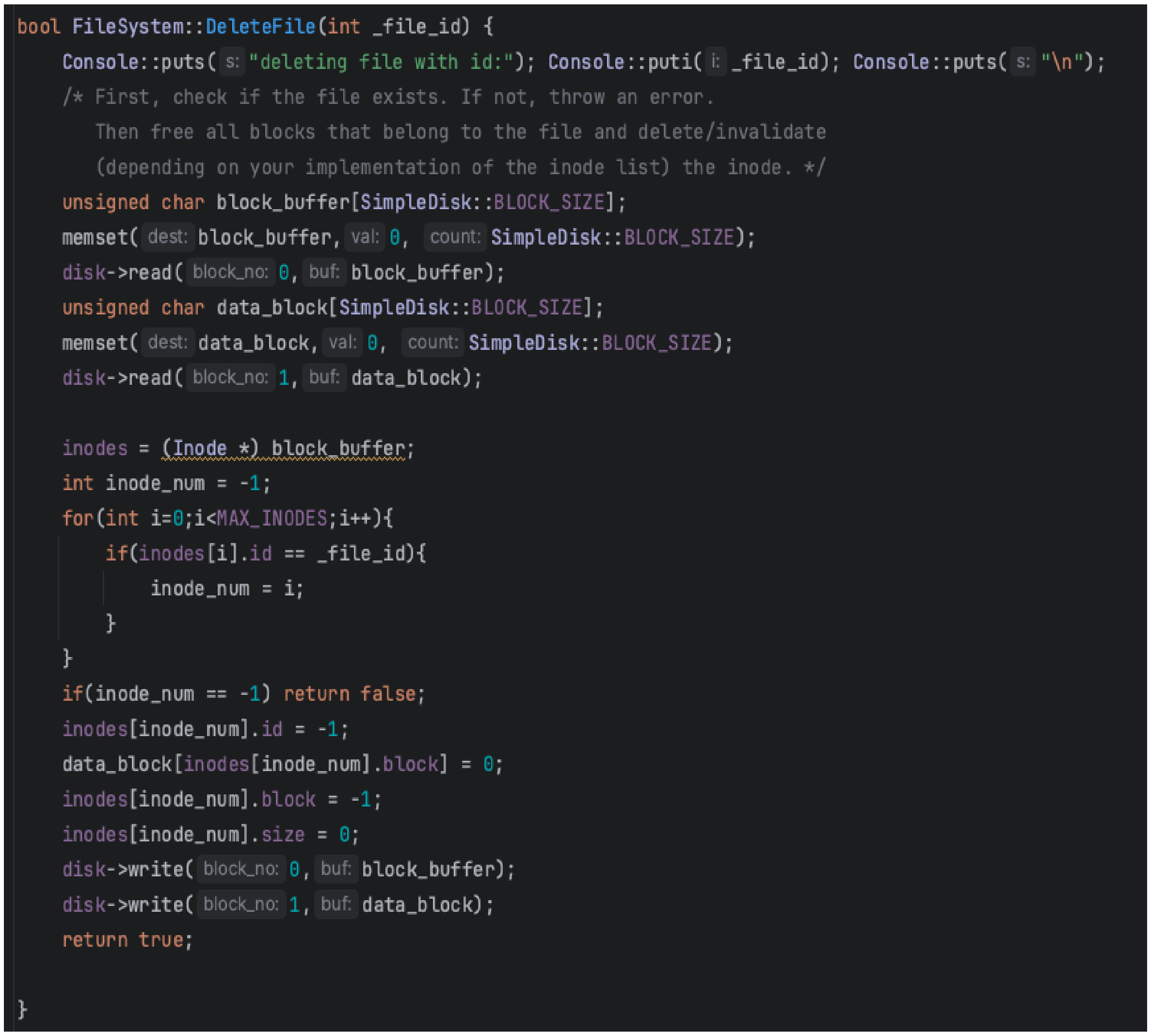
```
FileSystem::FileSystem() {
    Console::puts( s: "In file system constructor.\n");
    inodes = new Inode[SimpleDisk::BLOCK_SIZE];
    free_blocks = new unsigned char [SimpleDisk::BLOCK_SIZE];
}

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


/*------------------------------------------------------------------
/* FILE SYSTEM FUNCTIONS */
/*------------------------------------------------------------------


bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts( s: "mounting file system from disk\n");
    disk = _disk;
    /* Here you read the inode list and the free list into memory
    unsigned char block_buffer[SimpleDisk::BLOCK_SIZE];
    memset( dest: block_buffer, val: 0, count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 0, buf: block_buffer);
    inodes = (Inode*) block_buffer;
    memset( dest: block_buffer, val: 0, count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 1, buf: block_buffer);
    free_blocks = (unsigned char *) block_buffer;
    return true;
}
```

Figure 1: FileSystem Class Mount



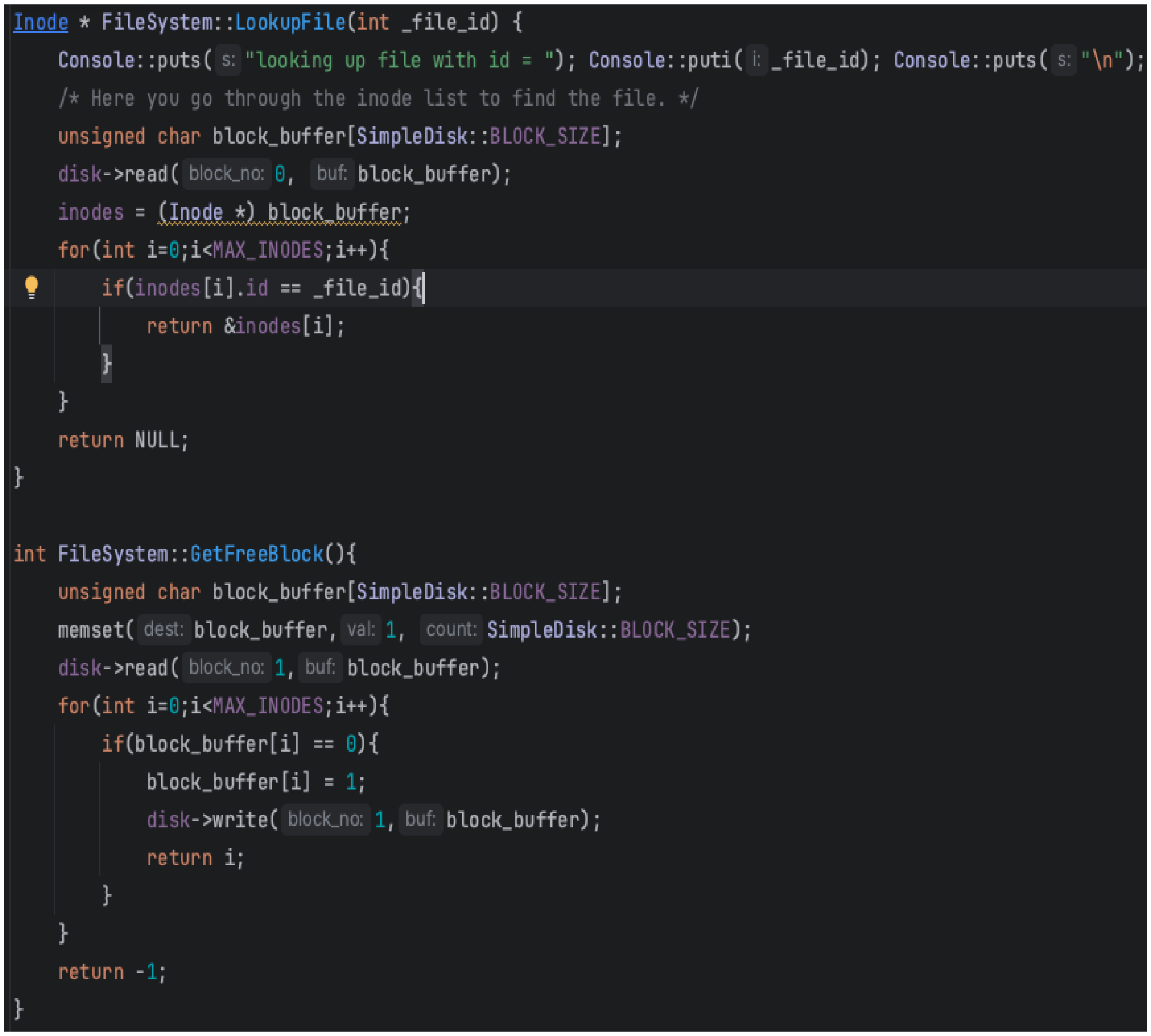
```
bool FileSystem::CreateFile(int _file_id) {
    Console::puts( s: "creating file with id:"); Console::puti( i: _file_id); Console::puts( s: "\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */
    unsigned char block_buffer[SimpleDisk::BLOCK_SIZE];
    memset( dest: block_buffer, val: 0,  count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 0, buf: block_buffer);
    inodes = (Inode *) block_buffer;
    int free_inode = -1;
    for(int i=0;i<MAX_INODES;i++){
        if(inodes[i].id == -1){
            free_inode = i;
        }
        if(inodes[i].id == _file_id){
            return false;
        }
    }
    inodes[free_inode].id = _file_id;
    inodes[free_inode].size = SimpleDisk::BLOCK_SIZE;
    inodes[free_inode].block = GetFreeBlock();
    disk->write( block_no: 0, buf: block_buffer);
    return true;


}
```

Figure 2: FileSystem Class Create



```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts( s: "deleting file with id:"); Console::puti( i: _file_id); Console::puts( s: "\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. */
    unsigned char block_buffer[SimpleDisk::BLOCK_SIZE];
    memset( dest: block_buffer, val: 0,  count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 0, buf: block_buffer);
    unsigned char data_block[SimpleDisk::BLOCK_SIZE];
    memset( dest: data_block, val: 0,  count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 1, buf: data_block);

    inodes = (Inode *) block_buffer;
    int inode_num = -1;
    for(int i=0;i<MAX_INODES;i++){
        if(inodes[i].id == _file_id){
            inode_num = i;
        }
    }
    if(inode_num == -1) return false;
    inodes[inode_num].id = -1;
    data_block[inodes[inode_num].block] = 0;
    inodes[inode_num].block = -1;
    inodes[inode_num].size = 0;
    disk->write( block_no: 0, buf: block_buffer);
    disk->write( block_no: 1, buf: data_block);
    return true;

}
```

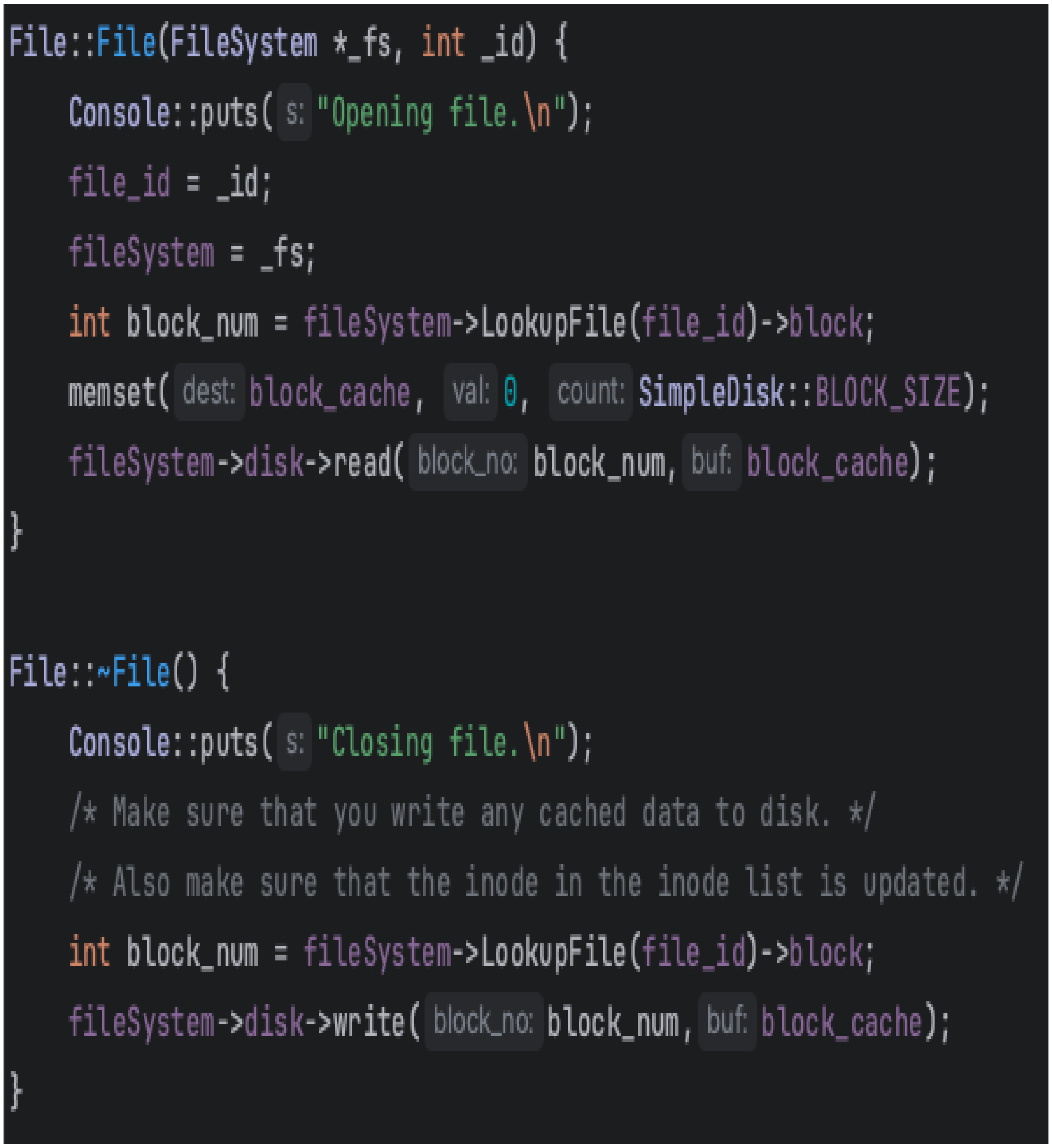
Figure 3: FileSystem Class Create



```cpp
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts( s: "looking up file with id = "); Console::puti( i: _file_id); Console::puts( s: "\n");
    /* Here you go through the inode list to find the file. */
    unsigned char block_buffer[SimpleDisk::BLOCK_SIZE];
    disk->read( block_no: 0,  buf: block_buffer);
    inodes = (Inode *) block_buffer;
    for(int i=0;i<MAX_INODES;i++){
        if(inodes[i].id == _file_id){
            return &inodes[i];
        }
    }
    return NULL;
}

int FileSystem::GetFreeBlock(){
    unsigned char block_buffer[SimpleDisk::BLOCK_SIZE];
    memset( dest: block_buffer, val: 1,  count: SimpleDisk::BLOCK_SIZE);
    disk->read( block_no: 1, buf: block_buffer);
    for(int i=0;i<MAX_INODES;i++){
        if(block_buffer[i] == 0){
            block_buffer[i] = 1;
            disk->write( block_no: 1, buf: block_buffer);
            return i;
        }
    }
    return -1;
}
```

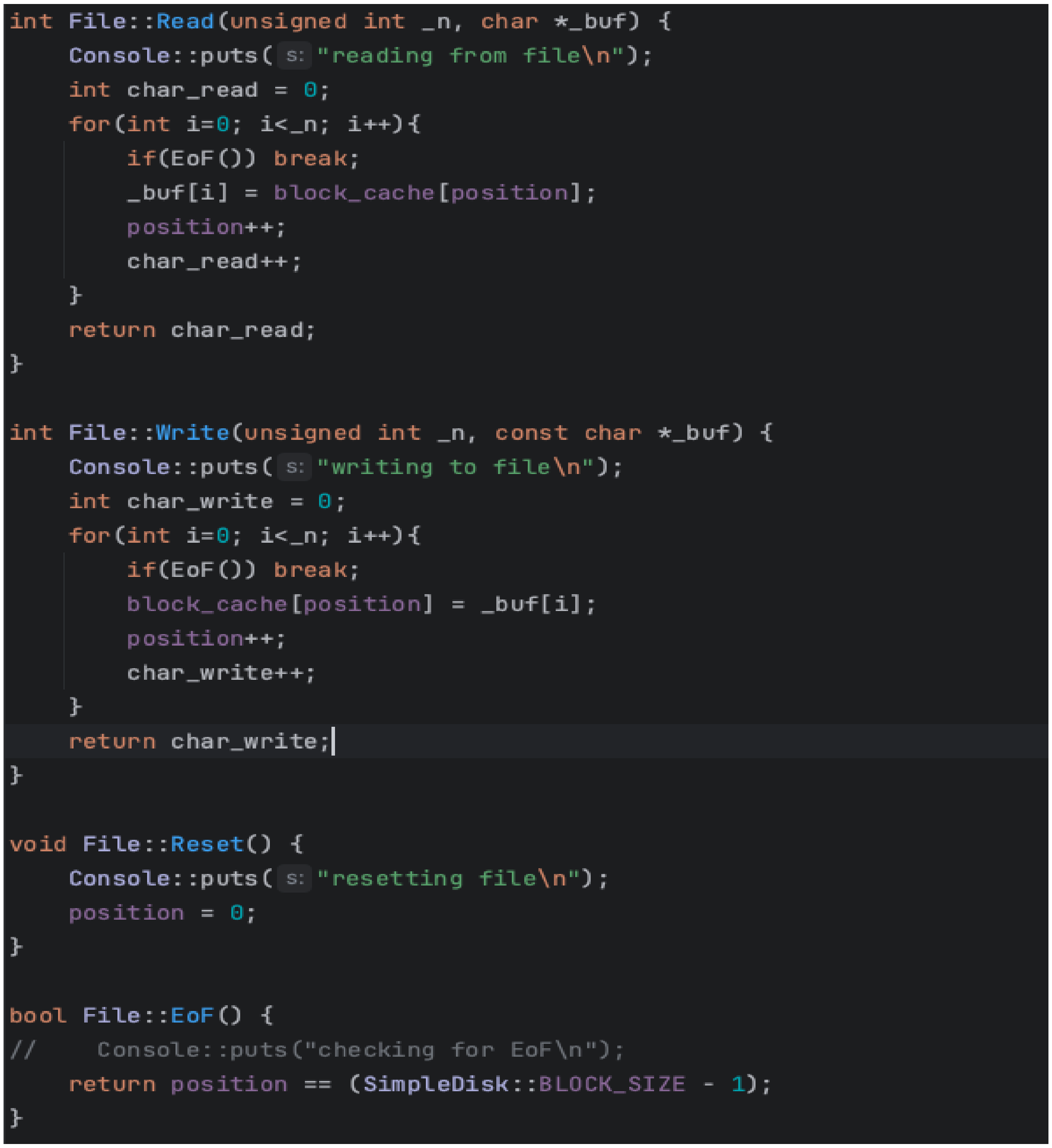
Figure 4: FileSystem Class Create

File Class :



```
File::File(FileSystem *_fs, int _id) {
    Console::puts( s: "Opening file.\n");
    file_id = _id;
    fileSystem = _fs;
    int block_num = fileSystem->LookupFile(file_id)->block;
    memset( dest: block_cache,  val: 0,  count: SimpleDisk::BLOCK_SIZE);
    fileSystem->disk->read( block_no: block_num,  buf: block_cache);
}


File::~File() {
    Console::puts( s: "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. */
    int block_num = fileSystem->LookupFile(file_id)->block;
    fileSystem->disk->write( block_no: block_num,  buf: block_cache);
}
```

Figure 5: File Class Constructor and Destructor

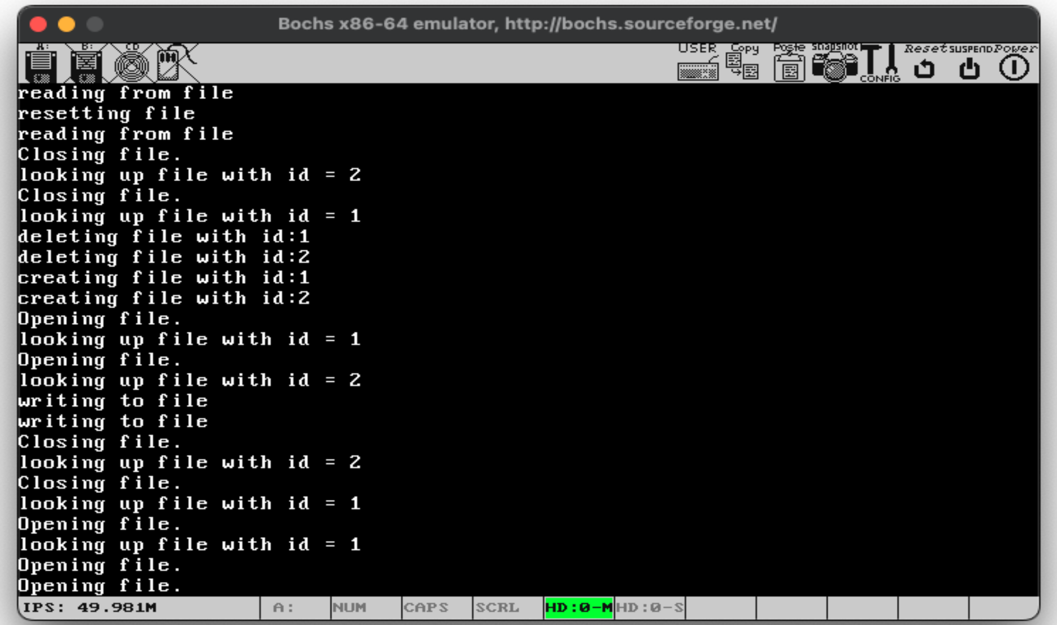


```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts( s: "reading from file\n");
    int char_read = 0;
    for(int i=0; i<_n; i++){
        if(EoF()) break;
        _buf[i] = block_cache[position];
        position++;
        char_read++;
    }
    return char_read;
}

int File::Write(unsigned int _n, const char *_buf) {
    Console::puts( s: "writing to file\n");
    int char_write = 0;
    for(int i=0; i<_n; i++){
        if(EoF()) break;
        block_cache[position] = _buf[i];
        position++;
        char_write++;
    }
    return char_write;
}

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

bool File::EoF() {
//    Console::puts("checking for EoF\n");
    return position == (SimpleDisk::BLOCK_SIZE - 1);
}
```

Figure 6: File Class Constructor and Destructor

Testing

For testing, I used the given test function from kernel.C. I ran the normal testing given in kernel. All the tests are passing, for given and additional scenarios.



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

Figure 7: Testing the creation, deletion, read and write two files with id 1 and 2.