MP7: Vanilla File System

Cheng-Yun Cheng
UIN: 633002216
CSCE611: Operating System

Assigned Tasks

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

System Design

The goal of the machine problem is to implement a file system and files which support sequential access only.

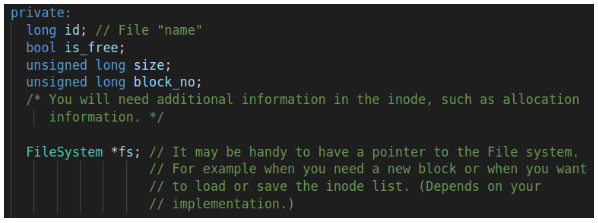
1. Main part:
   - The first data block and the second data block in disk are used for the inode list and the free list respectively.
   - An inode contains information including file name, file size, data block number, and used or not.
2. Option: Extend the file size to 64KB
   - Instead of having a data block directly, an inode has a index block which stores the data blocks of a file. (The size of a block is 512 and it can store 128 entries for data block numbers. Therefore, with a index block, the file size can be extend to 64KB.)
   - When creating a new file, two data blocks would be allocated, one for the index block and the other for the first data block.
   - When deleting a file, all the date blocks and the index block should be freed.
   - When reading a file, if the current block is finished, the next date block should be read into memory.
   - When writing a file, if the current block is finished, read the next block into memory or allocate a new data block for the file.

Code Description (Main)

For the main part, I modified the file_system.h, file_system.c, file.h, and file.c.

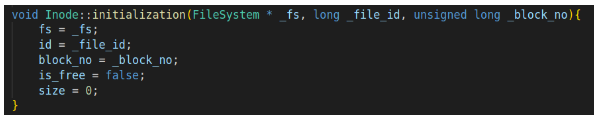
file_system.h: Inode Data Member :

- long id: File name
- bool is_free: Whether the inode is free or not
- unsigned long size: File size
- unsigned long block_no: Data block of the file
- FileSystem* fs: Pointer which point to the FileSyetem



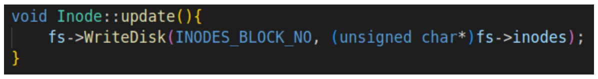
```
private:
  long id; // File "name"
  bool is_free;
  unsigned long size;
  unsigned long block_no;
  /* You will need additional information in the inode, such as allocation
     information. */

  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.)
```

file_system.c: Inode initialization : Initialize the inode. Set the local variable.



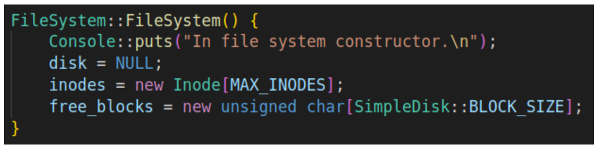
```
void Inode::initialization(FileSystem * _fs, long _file_id, unsigned long _block_no){
    fs = _fs;
    id = _file_id;
    block_no = _block_no;
    is_free = false;
    size = 0;
}
```

file_system.c: Inode update : Update the inode list into disk.



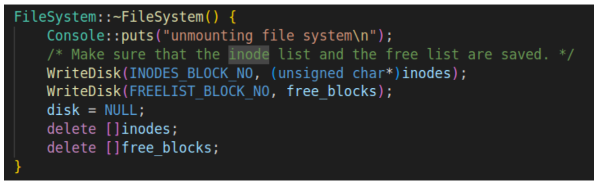
```
void Inode::update(){
    fs->WriteDisk(INODES_BLOCK_NO, (unsigned char*)fs->inodes);
}
```

file_system.c: FileSystem Constructor : Initialize the local data.



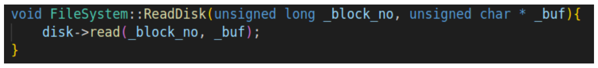
```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    disk = NULL;
    inodes = new Inode[MAX_INODES];
    free_blocks = new unsigned char[SimpleDisk::BLOCK_SIZE];
}
```

file_system.c: FileSystem Deconstructor : Unmount the file system and update the inode list and the free list into disk.



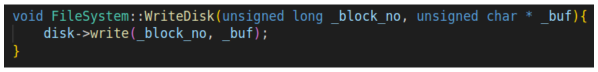
```
FileSystem::~FileSystem() {
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */
    WriteDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
    WriteDisk(FREELIST_BLOCK_NO, free_blocks);
    disk = NULL;
    delete []inodes;
    delete []free_blocks;
}
```

file_system.c: FileSystem ReadDisk : Call the read function in SimpleDisk.



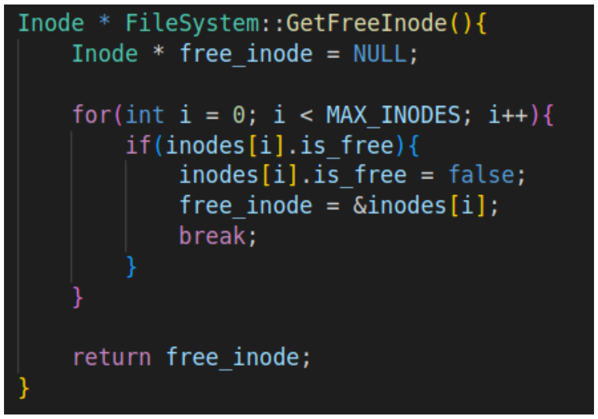
```
void FileSystem::ReadDisk(unsigned long _block_no, unsigned char * _buf){
    disk->read(_block_no, _buf);
}
```

file_system.c: FileSystem WriteDisk : Call the write function in SimpleDisk.



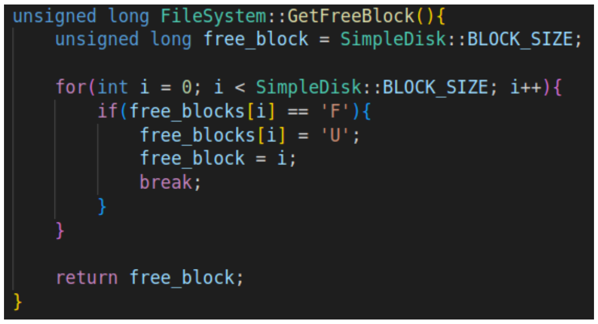
```
void FileSystem::WriteDisk(unsigned long _block_no, unsigned char * _buf){
    disk->write(_block_no, _buf);
}
```

file_system.c: FileSystem GetFreeInode : If there is a free node, set it as used and return it.



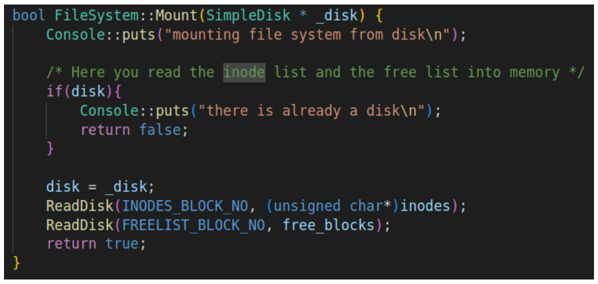
```
Inode * FileSystem::GetFreeInode(){
    Inode * free_inode = NULL;

    for(int i = 0; i < MAX_INODES; i++){
        if(inodes[i].is_free){
            inodes[i].is_free = false;
            free_inode = &inodes[i];
            break;
        }
    }

    return free_inode;
}
```

file_system.c: FileSystem GetFreeBlock : If there is a free block, set it as used and return it.



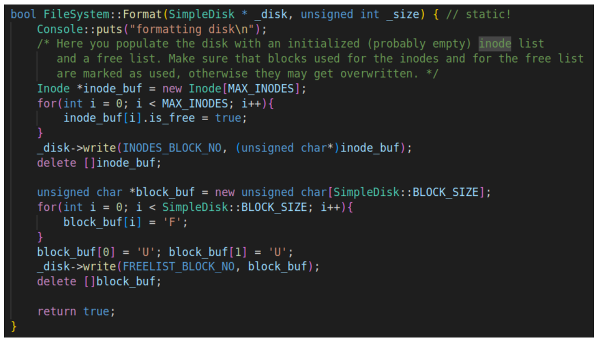
```
unsigned long FileSystem::GetFreeBlock(){
    unsigned long free_block = SimpleDisk::BLOCK_SIZE;

    for(int i = 0; i < SimpleDisk::BLOCK_SIZE; i++){
        if(free_blocks[i] == 'F'){
            free_blocks[i] = 'U';
            free_block = i;
            break;
        }
    }

    return free_block;
}
```

file_system.c: FileSystem Mount : Associate the file system with a disk, and read the inode list and free list from the disk.



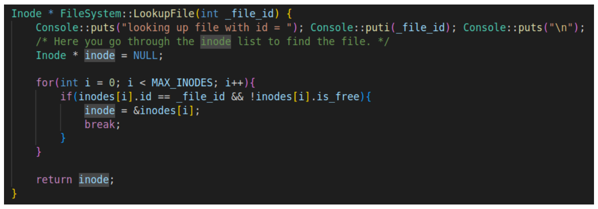
```
bool FileSystem::Mount(SimpleDisk * _disk) {
    Console::puts("mounting file system from disk\n");

    /* Here you read the inode list and the free list into memory */
    if(disk){
        Console::puts("there is already a disk\n");
        return false;
    }

    disk = _disk;
    ReadDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
    ReadDisk(FREELIST_BLOCK_NO, free_blocks);
    return true;
}
```

file_system.c: FileSystem Format : Initialize the inode list and free list into disk. Set all inodes and all data blocks as free. Only the first block and the second one set as used (for inode list and free list).



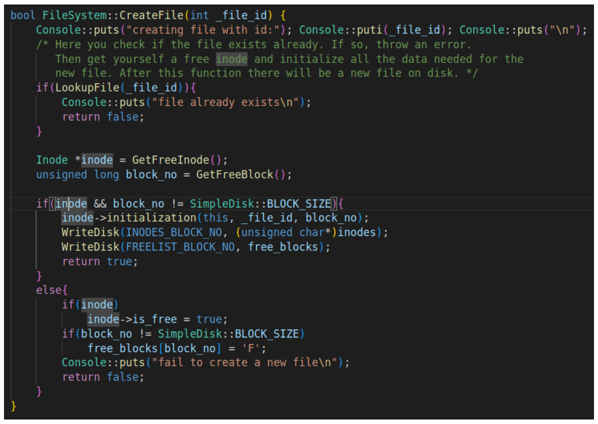
```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { // static!
    Console::puts("formatting disk\n");
    /* Here you populate the disk with an initialized (probably empty) inode list
       and a free list. Make sure that blocks used for the inodes and for the free list
       are marked as used, otherwise they may get overwritten. */
    Inode *inode_buf = new Inode[MAX_INODES];
    for(int i = 0; i < MAX_INODES; i++){
        inode_buf[i].is_free = true;
    }
    _disk->write(INODES_BLOCK_NO, (unsigned char*)inode_buf);
    delete []inode_buf;

    unsigned char *block_buf = new unsigned char[SimpleDisk::BLOCK_SIZE];
    for(int i = 0; i < SimpleDisk::BLOCK_SIZE; i++){
        block_buf[i] = 'F';
    }
    block_buf[0] = 'U'; block_buf[1] = 'U';
    _disk->write(FREELIST_BLOCK_NO, block_buf);
    delete []block_buf;

    return true;
}
```

file_system.c: FileSystem LookupFile : Go through the inode list to find a file.



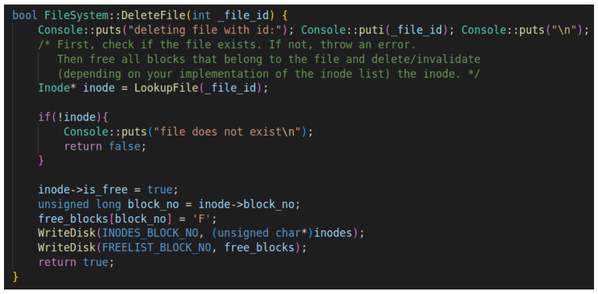
```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::puti(_file_id); Console::puts("\n");
    /* Here you go through the inode list to find the file. */
    Inode * inode = NULL;

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

    return inode;
}
```

file_system.c: FileSystem CreateFile : If the file name does not exist, create a new file. Get a free inode and a free block. Initialize the inode, and then update the inode list and free list into 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. */
    if(LookupFile(_file_id)){
        Console::puts("file already exists\n");
        return false;
    }

    Inode *inode = GetFreeInode();
    unsigned long block_no = GetFreeBlock();

    if(inode && block_no != SimpleDisk::BLOCK_SIZE){
        inode->initialization(this, _file_id, block_no);
        WriteDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
        WriteDisk(FREELIST_BLOCK_NO, free_blocks);
        return true;
    }
    else{
        if(inode)
            inode->is_free = true;
        if(block_no != SimpleDisk::BLOCK_SIZE)
            free_blocks[block_no] = 'F';
        Console::puts("fail to create a new file\n");
        return false;
    }
}
```

file_system.c: FileSystem DeleteFile : If the file name exist, delete this file. Set the inode and the data block as free, and then update the inode list and the free list into disk.

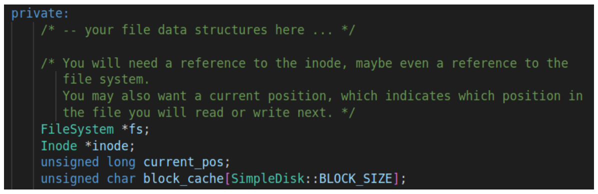


```
bool FileSystem::DeleteFile(int _file_id) {
    Console::puts("deleting file with id:"); Console::puti(_file_id); Console::puts("\n");
    /* First, check if the file exists. If not, throw an error.
       Then free all blocks that belong to the file and delete/invalidate
       (depending on your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

    if(!inode){
        Console::puts("file does not exist\n");
        return false;
    }

    inode->is_free = true;
    unsigned long block_no = inode->block_no;
    free_blocks[block_no] = 'F';
    WriteDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
    WriteDisk(FREELIST_BLOCK_NO, free_blocks);
    return true;
}
```

file.h: File Data Member :

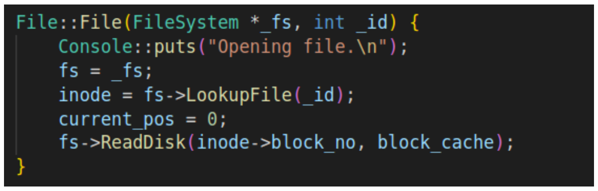
- FileSystem* fs: A pointer which points to FileSystem

- Inode* inode: A pointer which points to the inode
- unsigned long current_pos: The position of file that would be read or written
- unsigned char block_cache: The block storing the file data



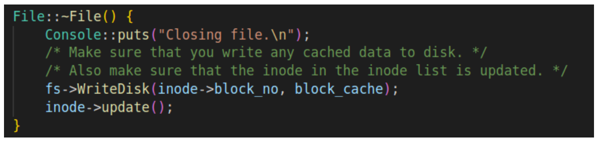
```
private:
    /* -- your file data structures here ... */

    /* 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. */
    FileSystem *fs;
    Inode *inode;
    unsigned long current_pos;
    unsigned char block_cache[SimpleDisk::BLOCK_SIZE];
```

file.c: Constructor : Initialize the local variable. Get the inode from the disk and read the date into memory.



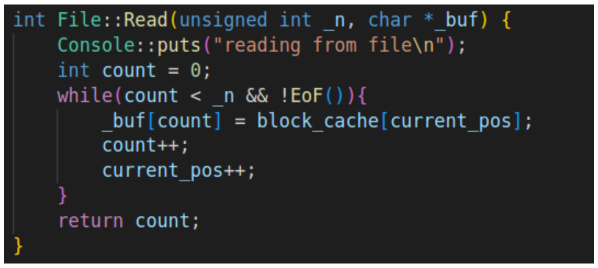
```
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file.\n");
    fs = _fs;
    inode = fs->LookupFile(_id);
    current_pos = 0;
    fs->ReadDisk(inode->block_no, block_cache);
}
```

file.c: Deconstructor : Write the data and update the inode into disk



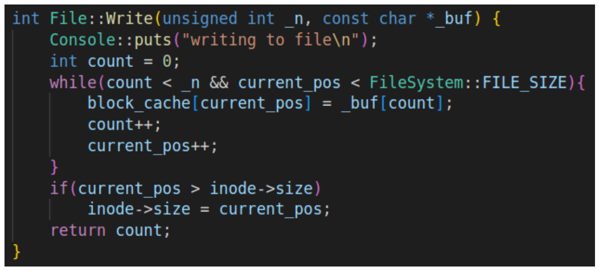
```
File::~File() {
    Console::puts("Closing file.\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */
    fs->WriteDisk(inode->block_no, block_cache);
    inode->update();
}
```

file.c: Read : Read data from the current position. End of the file should be checked to avoid from reading beyond the file.



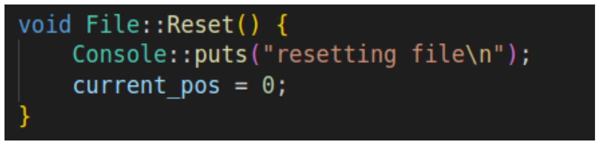
```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    int count = 0;
    while(count < _n && !EoF()){
        _buf[count] = block_cache[current_pos];
        count++;
        current_pos++;
    }
    return count;
}
```

file.c: Write : Write data to file starting at the current position. If the file size is extended, update the file size. Also, maximum file size should be check to avoid from writing beyond a data block.



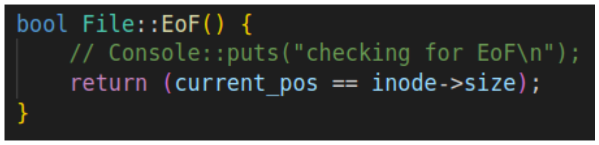
```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    int count = 0;
    while(count < _n && current_pos < FileSystem::FILE_SIZE){
        block_cache[current_pos] = _buf[count];
        count++;
        current_pos++;
    }
    if(current_pos > inode->size)
        inode->size = current_pos;
    return count;
}
```

file.c: Reset : Set the current position to the beginning of the file.



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

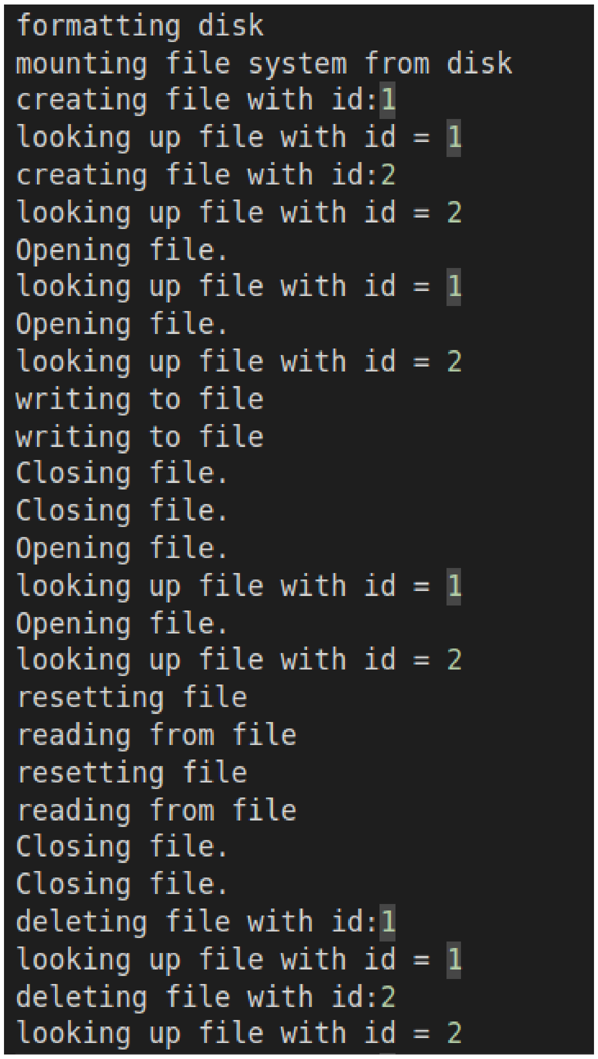
file.c: EoF : Whether the current position for the file is at the end of the file or not



```
bool File::EoF() {
    // Console::puts("checking for EoF\n");
    return (current_pos == inode->size);
}
```

Testing (Main)

For the machine problem, I only use the provided test.



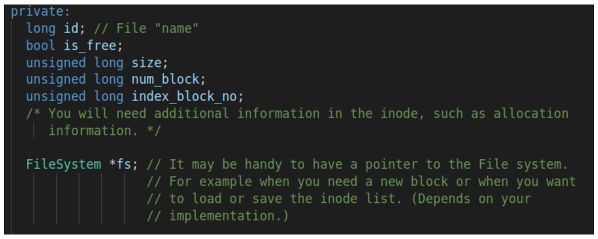
```
formatting disk
mounting file system from disk
creating file with id:1
looking up file with id = 1
creating file with id:2
looking up file with id = 2
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
writing to file
writing to file
Closing file.
Closing file.
Opening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

Code Description (Option)

For the option part, I modified the file_system.h, file_system.c, file.h, and file.c.

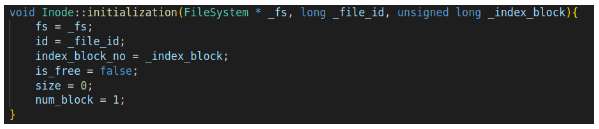
file_system.h: Inode Data Member :

- unsigned long num_block: Number of data blocks of the file
- unsigned long index_block_no: The index block



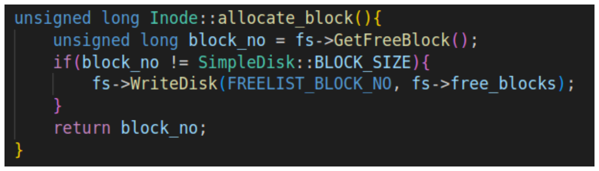
```
private:
  long id; // File "name"
  bool is_free;
  unsigned long size;
  unsigned long num_block;
  unsigned long index_block_no;
  /* You will need additional information in the inode, such as allocation
     information. */

  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.)
```

file_system.c: Inode initialization : Initialize the inode. Set the local variable.



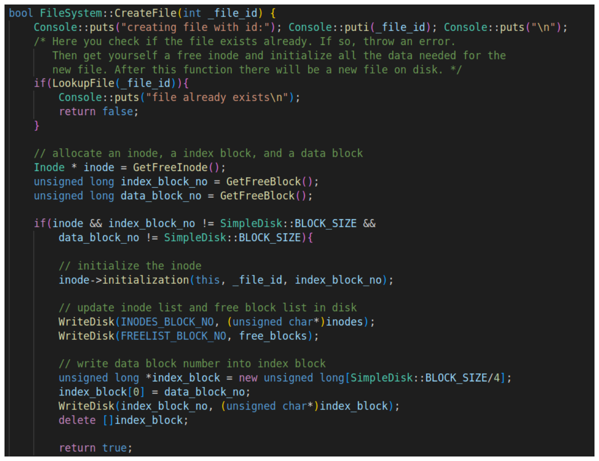
```
void Inode::initialization(FileSystem * _fs, long _file_id, unsigned long _index_block){
    fs = _fs;
    id = _file_id;
    index_block_no = _index_block;
    is_free = false;
    size = 0;
    num_block = 1;
}
```

file_system.c: Inode allocate_block : Return a free block.



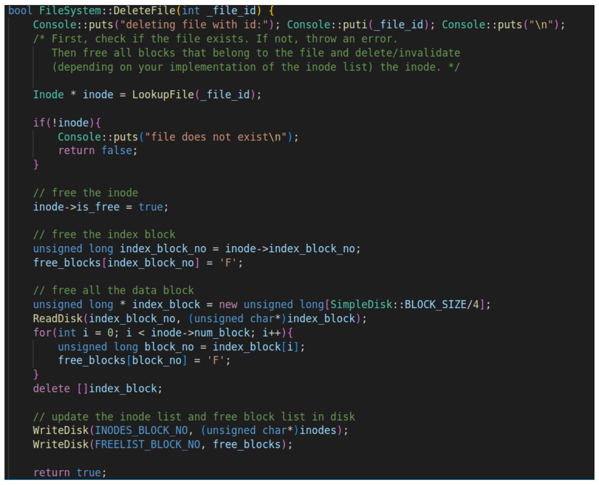
```
unsigned long Inode::allocate_block(){
    unsigned long block_no = fs->GetFreeBlock();
    if(block_no != SimpleDisk::BLOCK_SIZE){
        fs->WriteDisk(FREELIST_BLOCK_NO, fs->free_blocks);
    }
    return block_no;
}
```

file_system.c: FileSystem CreateFile : If the file name does not exist, create a new file. Get a free inode and two free block, one for the index block and the other for data block. Initialize the inode, and then update the inode list, free list, and the index block into 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. */
    if(LookupFile(_file_id)){
        Console::puts("file already exists\n");
        return false;
    }

    // allocate an inode, a index block, and a data block
    Inode * inode = GetFreeInode();
    unsigned long index_block_no = GetFreeBlock();
    unsigned long data_block_no = GetFreeBlock();

    if(inode && index_block_no != SimpleDisk::BLOCK_SIZE &&
        data_block_no != SimpleDisk::BLOCK_SIZE){

        // initialize the inode
        inode->initialization(this, _file_id, index_block_no);

        // update inode list and free block list in disk
        WriteDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
        WriteDisk(FREELIST_BLOCK_NO, free_blocks);

        // write data block number into index block
        unsigned long *index_block = new unsigned long[SimpleDisk::BLOCK_SIZE/4];
        index_block[0] = data_block_no;
        WriteDisk(index_block_no, (unsigned char*)index_block);
        delete []index_block;

        return true;
```

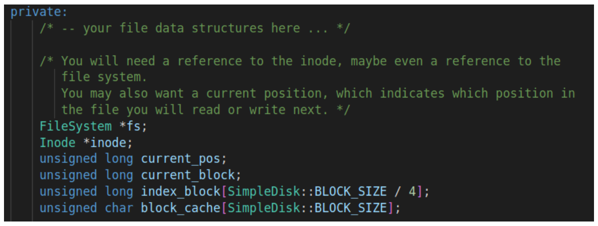
file_system.c: FileSystem DeleteFile : If the file name exist, delete this file. Set the inode, the index block, and all data blocks as free, and then update the inode list and the free list into disk.



```
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){
        Console::puts("file does not exist\n");
        return false;
    }

    // free the inode
    inode->is_free = true;

    // free the index block
    unsigned long index_block_no = inode->index_block_no;
    free_blocks[index_block_no] = 'F';

    // free all the data block
    unsigned long * index_block = new unsigned long[SimpleDisk::BLOCK_SIZE/4];
    ReadDisk(index_block_no, (unsigned char*)index_block);
    for(int i = 0; i < inode->num_block; i++){
        unsigned long block_no = index_block[i];
        free_blocks[block_no] = 'F';
    }
    delete []index_block;

    // update the inode list and free block list in disk
    WriteDisk(INODES_BLOCK_NO, (unsigned char*)inodes);
    WriteDisk(FREELIST_BLOCK_NO, free_blocks);

    return true;
```

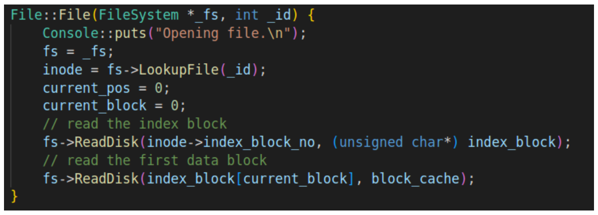
file.h: File Data Member :

- unsigned long current_block: The data block where the current position is.
- unsigned long index_block[]: The array storing all data block.



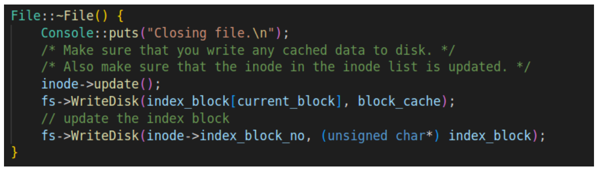
```
private:
    /* -- your file data structures here ... */

    /* 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. */
    FileSystem *fs;
    Inode *inode;
    unsigned long current_pos;
    unsigned long current_block;
    unsigned long index_block[SimpleDisk::BLOCK_SIZE / 4];
    unsigned char block_cache[SimpleDisk::BLOCK_SIZE];
```

file.c: Constructor : Initialize the local variable. Get the inode from the disk and read the index block and the first date block into memory.



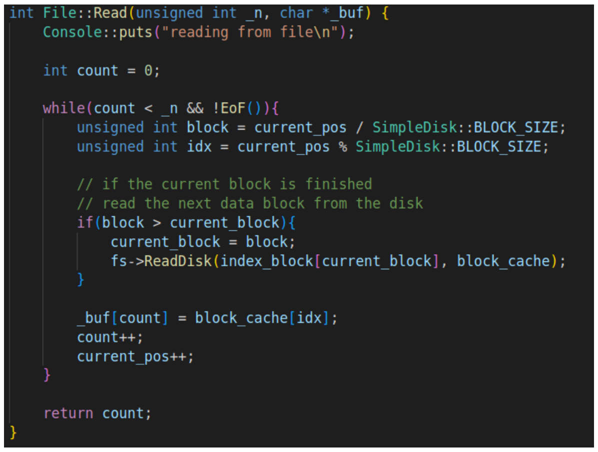
```
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file.\n");
    fs = _fs;
    inode = fs->LookupFile(_id);
    current_pos = 0;
    current_block = 0;
    // read the index block
    fs->ReadDisk(inode->index_block_no, (unsigned char*) index_block);
    // read the first data block
    fs->ReadDisk(index_block[current_block], block_cache);
}
```

file.c: Deconstructor : Write the data in the block_cache into disk. Update the inode and the index block into disk.



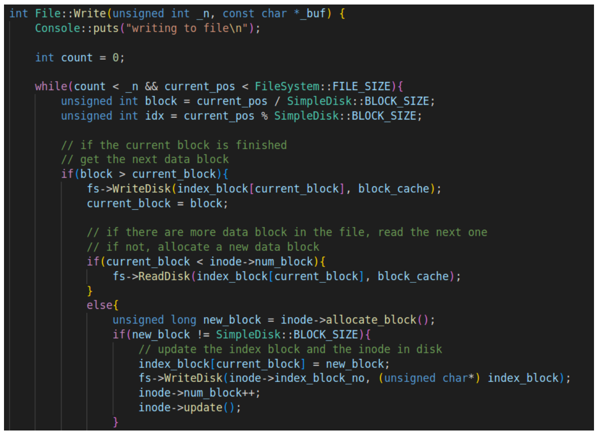
```
File::~File() {
    Console::puts("Closing file.\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */
    inode->update();
    fs->WriteDisk(index_block[current_block], block_cache);
    // update the index block
    fs->WriteDisk(inode->index_block_no, (unsigned char*) index_block);
}
```

file.c: Read : Read data from the current position. If the current data block is finished, read the next data block into memory.

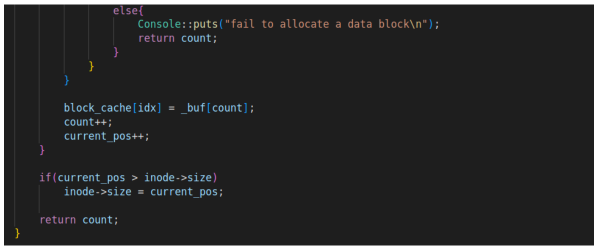


```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");

    int count = 0;

    while(count < _n && !EoF()){
        unsigned int block = current_pos / SimpleDisk::BLOCK_SIZE;
        unsigned int idx = current_pos % SimpleDisk::BLOCK_SIZE;

        // if the current block is finished
        // read the next data block from the disk
        if(block > current_block){
            current_block = block;
            fs->ReadDisk(index_block[current_block], block_cache);
        }

        _buf[count] = block_cache[idx];
        count++;
        current_pos++;
    }

    return count;
}
```

file.c: Write : Write data to file starting at the current position. If the current data block is full, write it into disk and read the next data block into memory. If the file doesn't have the next data block, allocate a new block. Then, update the index block and the inode into disk.

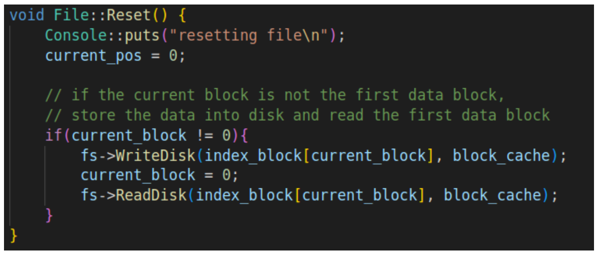


```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");

    int count = 0;

    while(count < _n && current_pos < FileSystem::FILE_SIZE){
        unsigned int block = current_pos / SimpleDisk::BLOCK_SIZE;
        unsigned int idx = current_pos % SimpleDisk::BLOCK_SIZE;

        // if the current block is finished
        // get the next data block
        if(block > current_block){
            fs->WriteDisk(index_block[current_block], block_cache);
            current_block = block;

            // if there are more data block in the file, read the next one
            // if not, allocate a new data block
            if(current_block < inode->num_block){
                fs->ReadDisk(index_block[current_block], block_cache);
            }
            else{
                unsigned long new_block = inode->allocate_block();
                if(new_block != SimpleDisk::BLOCK_SIZE){
                    // update the index block and the inode in disk
                    index_block[current_block] = new_block;
                    fs->WriteDisk(inode->index_block_no, (unsigned char*) index_block);
                    inode->num_block++;
                    inode->update();
                }
```



```
                else{
                    Console::puts("fail to allocate a data block\n");
                    return count;
                }
            }
        }

        block_cache[idx] = _buf[count];
        count++;
        current_pos++;
    }

    if(current_pos > inode->size)
        inode->size = current_pos;

    return count;
}
```

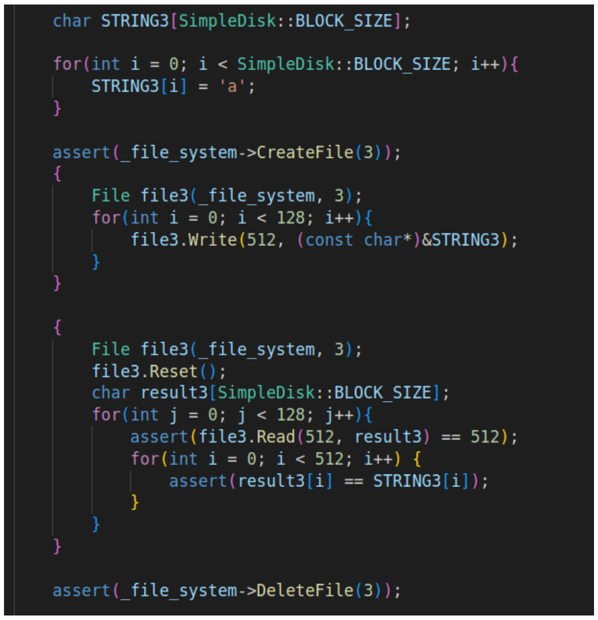
file.c: Reset : Set the current position to the beginning of the file. If the current data block is not the first one, write it into disk and read the first data block into memory.



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

    // if the current block is not the first data block,
    // store the data into disk and read the first data block
    if(current_block != 0){
        fs->WriteDisk(index_block[current_block], block_cache);
        current_block = 0;
        fs->ReadDisk(index_block[current_block], block_cache);
    }
}
```

Testing (Option)

To test the option part, I re-write the exercise_file_system function in kernel.c. All the process remains the same as the original function, except that 64KB data would be written into the file. The option to switch between the original and modified functions is controlled by the use of EXTEND_FILE_SIZE.



```
char STRING3[SimpleDisk::BLOCK_SIZE];

for(int i = 0; i < SimpleDisk::BLOCK_SIZE; i++){
    STRING3[i] = 'a';
}

assert(_file_system->CreateFile(3));
{
    File file3(_file_system, 3);
    for(int i = 0; i < 128; i++){
        file3.Write(512, (const char*)&STRING3);
    }
}

{
    File file3(_file_system, 3);
    file3.Reset();
    char result3[SimpleDisk::BLOCK_SIZE];
    for(int j = 0; j < 128; j++){
        assert(file3.Read(512, result3) == 512);
        for(int i = 0; i < 512; i++) {
            assert(result3[i] == STRING3[i]);
        }
    }
}

assert(_file_system->DeleteFile(3));
```