MP7: Simple File System

Ashutosh Punyani
UIN: 834006613
CSCE611: Operating System

Assigned Tasks

Main: Completed (Implementation of a simple file system that support sequential access)
Bonus Option 1: Completed (DESIGN of an extension to the basic file system to allow for files that are up to 64kB long)
Bonus Option 2: Completed (IMPLEMENTATION of the extensions proposed in Option 1)

System Design

The main objective of machine problem 7 is to develop a file system and files that exclusively support sequential access, catering to the context of a computer science undergraduate student. The primary components involve utilizing the first and second data blocks on the disk for the inode list and the free list, respectively. An inode, which encapsulates details such as file name, file size, data block number, and usage status, plays a pivotal role. As an additional option, there is a proposal to expand the file size to 64KB. This entails a modification where an inode, instead of directly containing data blocks, incorporates an index block to store the file's data blocks. With each block having a size of 512 and the capacity to store 128 entries for data block numbers, employing an index block facilitates extending the file size to 64KB. The file creation process involves allocating two data blocks: one for the index block and the other for the initial data block. Deleting a file requires freeing all data blocks and the index block. Reading a file mandates loading the next data block into memory when the current block is completed. Similarly, writing a file necessitates loading the next block into memory or allocating a new data block for the file when the current block is filled.

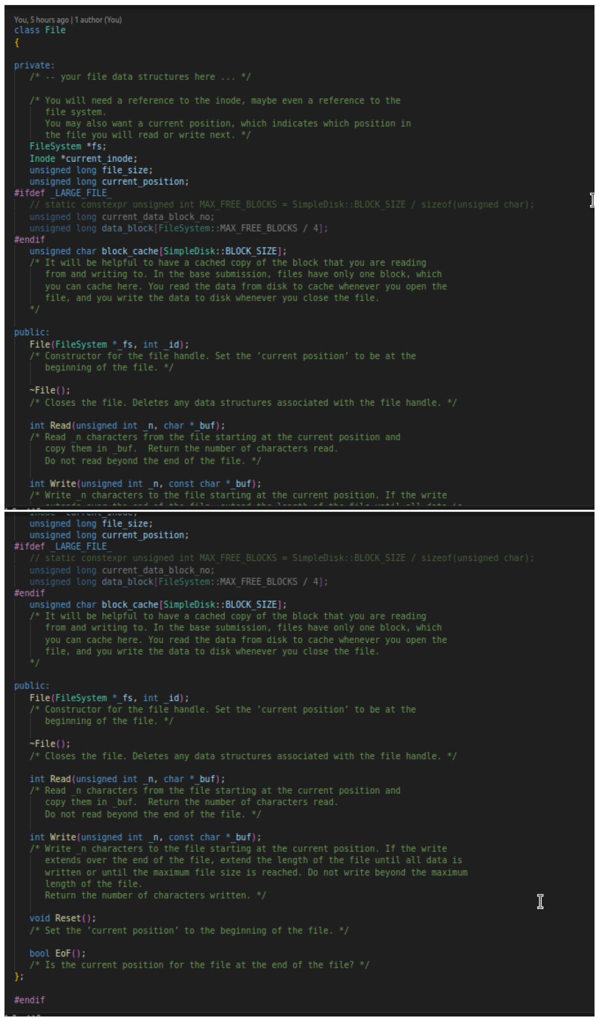
Code Description

During the implementation of this machine problem, I made changes to the following six files:

1. **file.H**
2. **file.C**
3. **file_system.H**
4. **file_system.C**
5. **kernel.C**

1. **file.H**

   In `file.H`, I have added the file data structures required for this implementation: a pointer that points to the FileSystem, a pointer that points to the inode, the position of the file to be read or written, the block storing the file data, the data block where the current position is, and the array storing all data blocks.



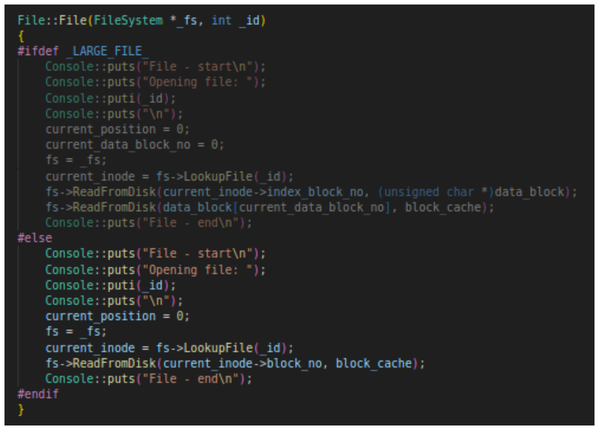
```
You, 5 hours ago | 1 author (You)
class File
{

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. */
   FileSystem *fs;
   Inode *current_inode;
   unsigned long file_size;
   unsigned long current_position;
#ifdef _LARGE_FILE_
   // static constexpr unsigned int MAX_FREE_BLOCKS = SimpleDisk::BLOCK_SIZE / sizeof(unsigned char);
   unsigned long current_data_block_no;
   unsigned long data_block[FileSystem::MAX_FREE_BLOCKS / 4];
#endif
   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.
   */

public:
   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
```

```
   unsigned long file_size;
   unsigned long current_position;
#ifdef _LARGE_FILE_
   // static constexpr unsigned int MAX_FREE_BLOCKS = SimpleDisk::BLOCK_SIZE / sizeof(unsigned char);
   unsigned long current_data_block_no;
   unsigned long data_block[FileSystem::MAX_FREE_BLOCKS / 4];
#endif
   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.
   */

public:
   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? */
};

#endif
```

Figure 1: **file.H**

2. **file.C**

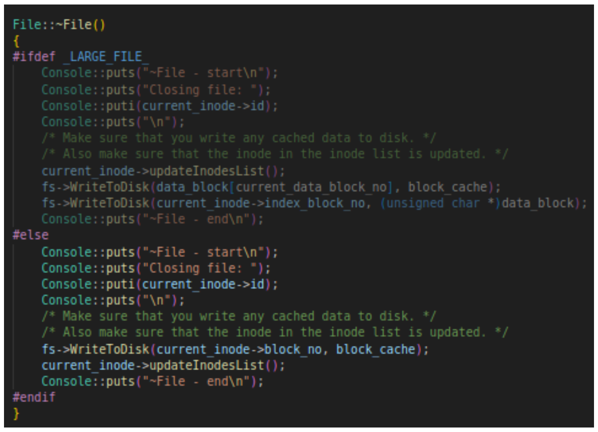
   (a) **File::File()** : Start by setting up the local variable. Then, fetch the inode from the disk, and load both the index block and the initial data block into the computer's memory.



```
File::File(FileSystem *_fs, int _id)
{
#ifdef _LARGE_FILE_
    Console::puts("File - start\n");
    Console::puts("Opening file: ");
    Console::puti(_id);
    Console::puts("\n");
    current_position = 0;
    current_data_block_no = 0;
    fs = _fs;
    current_inode = fs->LookupFile(_id);
    fs->ReadFromDisk(current_inode->index_block_no, (unsigned char *)data_block);
    fs->ReadFromDisk(data_block[current_data_block_no], block_cache);
    Console::puts("File - end\n");
#else
    Console::puts("File - start\n");
    Console::puts("Opening file: ");
    Console::puti(_id);
    Console::puts("\n");
    current_position = 0;
    fs = _fs;
    current_inode = fs->LookupFile(_id);
    fs->ReadFromDisk(current_inode->block_no, block_cache);
    Console::puts("File - end\n");
#endif
}
```

Figure 2: **File::File()**

   (b) **File:: File()** : Save the information stored in the block cache onto the disk. After that, make sure to update the inode and the index block on the disk to reflect any changes.

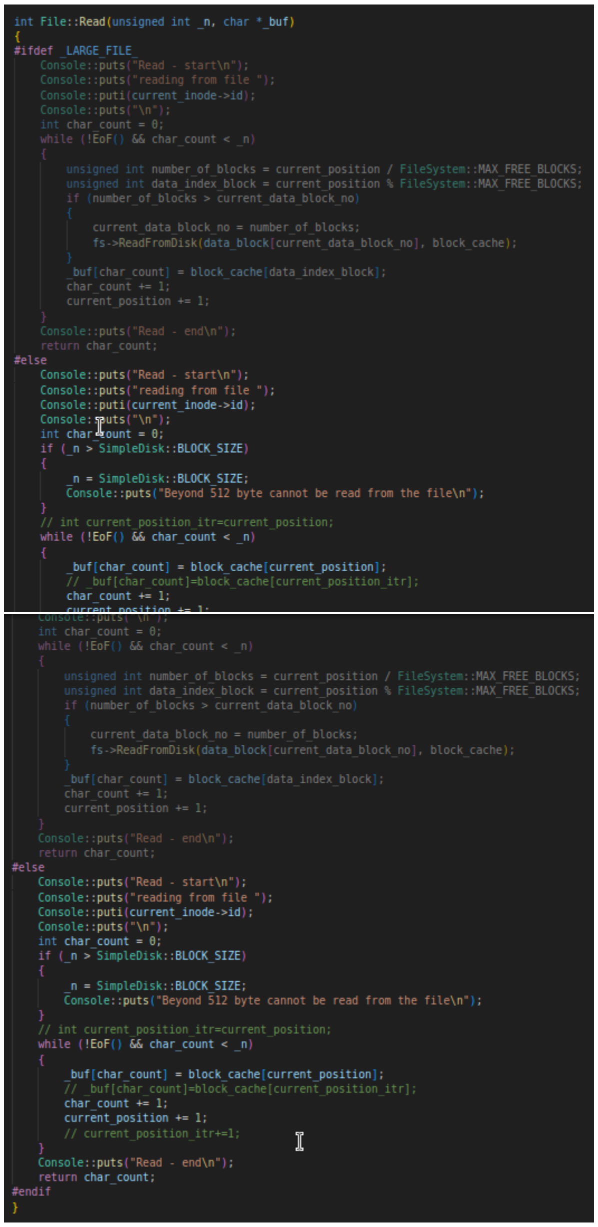


```
File::~File()
{
#ifdef _LARGE_FILE_
    Console::puts("~File - start\n");
    Console::puts("Closing file: ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */
    current_inode->updateInodesList();
    fs->WriteToDisk(data_block[current_data_block_no], block_cache);
    fs->WriteToDisk(current_inode->index_block_no, (unsigned char *)data_block);
    Console::puts("~File - end\n");
#else
    Console::puts("~File - start\n");
    Console::puts("Closing file: ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated. */
    fs->WriteToDisk(current_inode->block_no, block_cache);
    current_inode->updateInodesList();
    Console::puts("~File - end\n");
#endif
}
```

Figure 3: **File:: File()**

(c) **File::Read** :

Retrieve information from the current location in the file. If the file is small, make sure to check for the end of the file to prevent reading beyond its limits. For larger files, if the current data block is fully used up, read the next data block into the computer's memory.

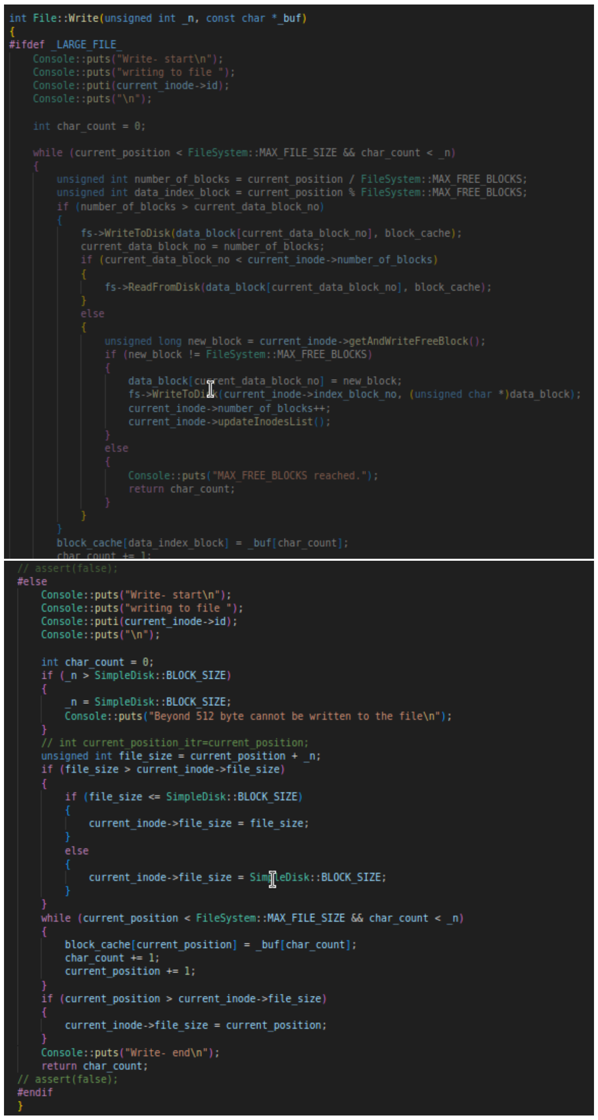


```
int File::Read(unsigned int _n, char *_buf)
{
#ifdef _LARGE_FILE_
    Console::puts("Read - start\n");
    Console::puts("reading from file ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    int char_count = 0;
    while (!EoF() && char_count < _n)
    {
        unsigned int number_of_blocks = current_position / FileSystem::MAX_FREE_BLOCKS;
        unsigned int data_index_block = current_position % FileSystem::MAX_FREE_BLOCKS;
        if (number_of_blocks > current_data_block_no)
        {
            current_data_block_no = number_of_blocks;
            fs->ReadFromDisk(data_block[current_data_block_no], block_cache);
        }
        _buf[char_count] = block_cache[data_index_block];
        char_count += 1;
        current_position += 1;
    }
    Console::puts("Read - end\n");
    return char_count;
#else
    Console::puts("Read - start\n");
    Console::puts("reading from file ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    int char_count = 0;
    if (_n > SimpleDisk::BLOCK_SIZE)
    {
        _n = SimpleDisk::BLOCK_SIZE;
        Console::puts("Beyond 512 byte cannot be read from the file\n");
    }
    // int current_position_itr=current_position;
    while (!EoF() && char_count < _n)
    {
        _buf[char_count] = block_cache[current_position];
        // _buf[char_count]=block_cache[current_position_itr];
        char_count += 1;
        current_position += 1;
```

```
    Console::puts("\n");
    int char_count = 0;
    while (!EoF() && char_count < _n)
    {
        unsigned int number_of_blocks = current_position / FileSystem::MAX_FREE_BLOCKS;
        unsigned int data_index_block = current_position % FileSystem::MAX_FREE_BLOCKS;
        if (number_of_blocks > current_data_block_no)
        {
            current_data_block_no = number_of_blocks;
            fs->ReadFromDisk(data_block[current_data_block_no], block_cache);
        }
        _buf[char_count] = block_cache[data_index_block];
        char_count += 1;
        current_position += 1;
    }
    Console::puts("Read - end\n");
    return char_count;
#else
    Console::puts("Read - start\n");
    Console::puts("reading from file ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    int char_count = 0;
    if (_n > SimpleDisk::BLOCK_SIZE)
    {
        _n = SimpleDisk::BLOCK_SIZE;
        Console::puts("Beyond 512 byte cannot be read from the file\n");
    }
    // int current_position_itr=current_position;
    while (!EoF() && char_count < _n)
    {
        _buf[char_count] = block_cache[current_position];
        // _buf[char_count]=block_cache[current_position_itr];
        char_count += 1;
        current_position += 1;
        // current_position_itr+=1;
    }
    Console::puts("Read - end\n");
    return char_count;
#endif
}
```

Figure 4: **File::Read**

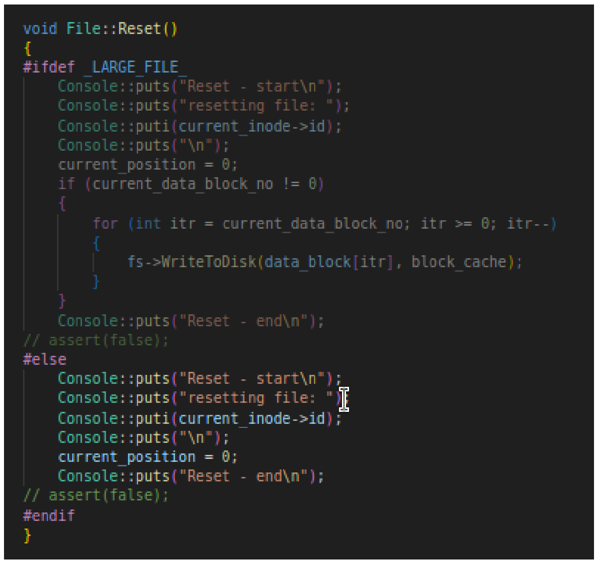
(d) **File::Write)** : Save information to a file, starting from the current position. For smaller files, if the file size increases, make sure to update the file size. Also, check the maximum file size to avoid writing beyond a data block. For larger files, if the current data block is full, save it to the disk and load the next data block into the computer's memory. If there's no next data block, create a new one. Then, update both the index block and the inode on the disk. If you're working with a file in C programming language and want to reset to the beginning, set the current position accordingly. If the current data block isn't the first one, store it on the disk and load the first data block into memory.



```cpp
int File::Write(unsigned int _n, const char *_buf)
{
#ifdef _LARGE_FILE_
    Console::puts("Write- start\n");
    Console::puts("writing to file ");
    Console::puti(current_inode->id);
    Console::puts("\n");

    int char_count = 0;

    while (current_position < FileSystem::MAX_FILE_SIZE && char_count < _n)
    {
        unsigned int number_of_blocks = current_position / FileSystem::MAX_FREE_BLOCKS;
        unsigned int data_index_block = current_position % FileSystem::MAX_FREE_BLOCKS;
        if (number_of_blocks > current_data_block_no)
        {
            fs->WriteToDisk(data_block[current_data_block_no], block_cache);
            current_data_block_no = number_of_blocks;
            if (current_data_block_no < current_inode->number_of_blocks)
            {
                fs->ReadFromDisk(data_block[current_data_block_no], block_cache);
            }
            else
            {
                unsigned long new_block = current_inode->getAndWriteFreeBlock();
                if (new_block != FileSystem::MAX_FREE_BLOCKS)
                {
                    data_block[current_data_block_no] = new_block;
                    fs->WriteToDisk(current_inode->index_block_no, (unsigned char *)data_block);
                    current_inode->number_of_blocks++;
                    current_inode->updateInodesList();
                }
                else
                {
                    Console::puts("MAX_FREE_BLOCKS reached.");
                    return char_count;
                }
            }
        }
        block_cache[data_index_block] = _buf[char_count];
        char_count += 1;
// assert(false);
#else
    Console::puts("Write- start\n");
    Console::puts("writing to file ");
    Console::puti(current_inode->id);
    Console::puts("\n");

    int char_count = 0;
    if (_n > SimpleDisk::BLOCK_SIZE)
    {
        _n = SimpleDisk::BLOCK_SIZE;
        Console::puts("Beyond 512 byte cannot be written to the file\n");
    }
    // int current_position_itr=current_position;
    unsigned int file_size = current_position + _n;
    if (file_size > current_inode->file_size)
    {
        if (file_size <= SimpleDisk::BLOCK_SIZE)
        {
            current_inode->file_size = file_size;
        }
        else
        {
            current_inode->file_size = SimpleDisk::BLOCK_SIZE;
        }
    }
    while (current_position < FileSystem::MAX_FILE_SIZE && char_count < _n)
    {
        block_cache[current_position] = _buf[char_count];
        char_count += 1;
        current_position += 1;
    }
    if (current_position > current_inode->file_size)
    {
        current_inode->file_size = current_position;
    }
    Console::puts("Write- end\n");
    return char_count;
// assert(false);
#endif
}
```

Figure 5: **File::Write)**

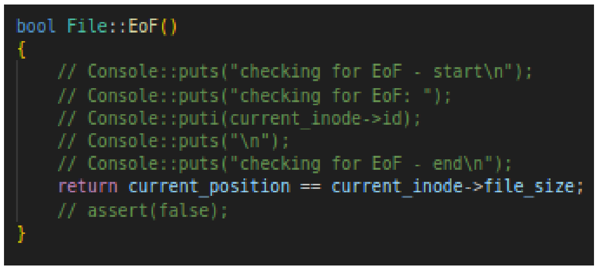
(e) **File::Reset()** : For small files, move the current position back to the start of the file. For larger files, also set the current position to the beginning. If the current data block is not the initial one, store it on the disk, and then load the first data block into the computer's memory.



```
void File::Reset()
{
#ifdef _LARGE_FILE_
    Console::puts("Reset - start\n");
    Console::puts("resetting file: ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    current_position = 0;
    if (current_data_block_no != 0)
    {
        for (int itr = current_data_block_no; itr >= 0; itr--)
        {
            fs->WriteToDisk(data_block[itr], block_cache);
        }
    }
    Console::puts("Reset - end\n");
// assert(false);
#else
    Console::puts("Reset - start\n");
    Console::puts("resetting file: ");
    Console::puti(current_inode->id);
    Console::puts("\n");
    current_position = 0;
    Console::puts("Reset - end\n");
// assert(false);
#endif
}
```

Figure 6: **File::Reset()**

(f) **File::EoF()** : For both small and large files, check if the current position in the file is at the end or not.

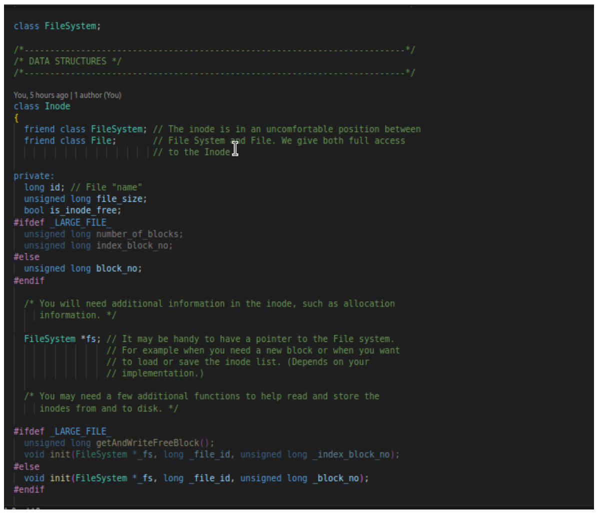


```
bool File::EoF()
{
    // Console::puts("checking for EoF - start\n");
    // Console::puts("checking for EoF: ");
    // Console::puti(current_inode->id);
    // Console::puts("\n");
    // Console::puts("checking for EoF - end\n");
    return current_position == current_inode->file_size;
    // assert(false);
}
```

Figure 7: **File::EoF()**

3. **file_system.H**

I have added the required data structures for the FileSystem in the file system: File name, Whether the inode is free or not, File size, Data block of the file, Pointer which points to the FileSystem, Number of data blocks of the file, The index block.



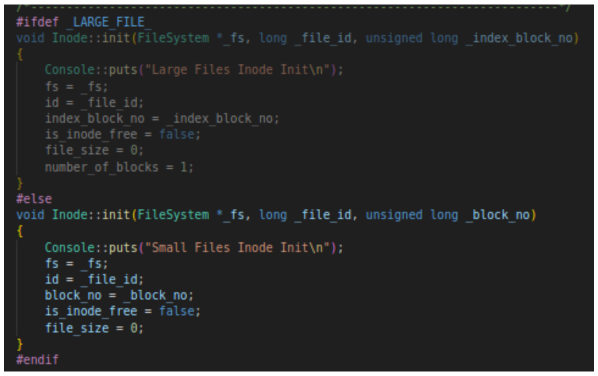
```
class FileSystem;

/*-----------------------------------------------------------------------*/
/* DATA STRUCTURES */
/*-----------------------------------------------------------------------*/

You, 5 hours ago | 1 author (You)
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"
  unsigned long file_size;
  bool is_inode_free;
#ifdef _LARGE_FILE_
  unsigned long number_of_blocks;
  unsigned long index_block_no;
#else
  unsigned long block_no;
#endif

  /* 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. */

#ifdef _LARGE_FILE_
  unsigned long getAndWriteFreeBlock();
  void init(FileSystem *_fs, long _file_id, unsigned long _index_block_no);
#else
  void init(FileSystem *_fs, long _file_id, unsigned long _block_no);
#endif
```

Figure 8: **file_system.H**

4. **file_system.C**

   (a) **Inode::init**

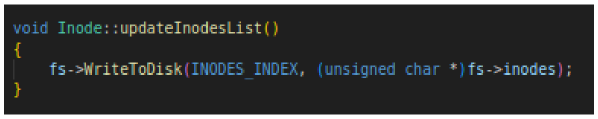
   For smaller files, start by setting up the inode and adjusting the local variable. Similarly, for larger files, initiate the inode and configure the local variable.



```
#ifdef _LARGE_FILE_
void Inode::init(FileSystem *_fs, long _file_id, unsigned long _index_block_no)
{
    Console::puts("Large Files Inode Init\n");
    fs = _fs;
    id = _file_id;
    index_block_no = _index_block_no;
    is_inode_free = false;
    file_size = 0;
    number_of_blocks = 1;
}
#else
void Inode::init(FileSystem *_fs, long _file_id, unsigned long _block_no)
{
    Console::puts("Small Files Inode Init\n");
    fs = _fs;
    id = _file_id;
    block_no = _block_no;
    is_inode_free = false;
    file_size = 0;
}
#endif
```

Figure 9: **Inode::init**

   (b) **Inode::updateInodesList()** For both small and large files, make sure to save the updated list of inodes onto the disk.

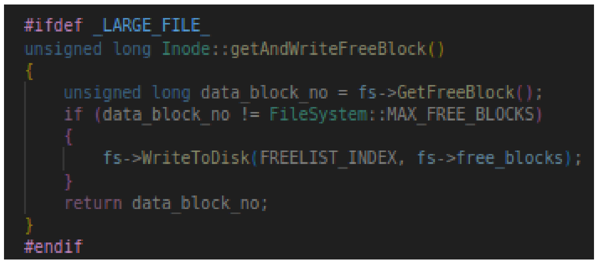


```
void Inode::updateInodesList()
{
    fs->WriteToDisk(INODES_INDEX, (unsigned char *)fs->inodes);
}
```

Figure 10: **Inode::updateInodesList()**

   (c) **Inode::getAndWriteFreeBlock()**

Specifically for larger files, provide a block that is available or free.

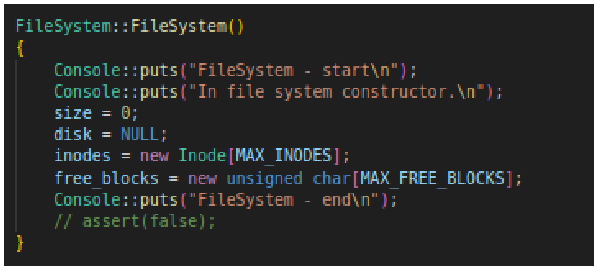


```
#ifdef _LARGE_FILE_
unsigned long Inode::getAndWriteFreeBlock()
{
    unsigned long data_block_no = fs->GetFreeBlock();
    if (data_block_no != FileSystem::MAX_FREE_BLOCKS)
    {
        fs->WriteToDisk(FREELIST_INDEX, fs->free_blocks);
    }
    return data_block_no;
}
#endif
```

Figure 11: **Inode::getAndWriteFreeBlock()**

(d) **FileSystem::FileSystem()**

For both small and large files, begin by setting up the local data.

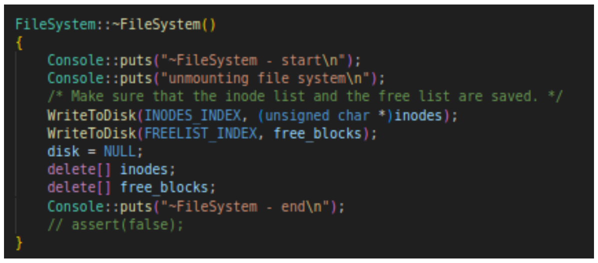


```
FileSystem::FileSystem()
{
    Console::puts("FileSystem - start\n");
    Console::puts("In file system constructor.\n");
    size = 0;
    disk = NULL;
    inodes = new Inode[MAX_INODES];
    free_blocks = new unsigned char[MAX_FREE_BLOCKS];
    Console::puts("FileSystem - end\n");
    // assert(false);
}
```

Figure 12: **FileSystem::FileSystem()**

(e) **FileSystem:: FileSystem()**

For both small and large files, dismount the file system, and save the updated lists of inodes and free space onto the disk.

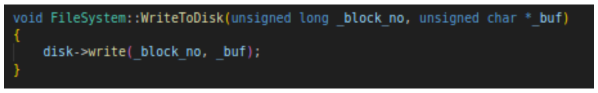


```
FileSystem::~FileSystem()
{
    Console::puts("~FileSystem - start\n");
    Console::puts("unmounting file system\n");
    /* Make sure that the inode list and the free list are saved. */
    WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
    WriteToDisk(FREELIST_INDEX, free_blocks);
    disk = NULL;
    delete[] inodes;
    delete[] free_blocks;
    Console::puts("~FileSystem - end\n");
    // assert(false);
}
```

Figure 13: **FileSystem:: FileSystem()**

(f) **FileSystem::WriteToDisk()**

For both small and large files, use the write function in SimpleDisk.

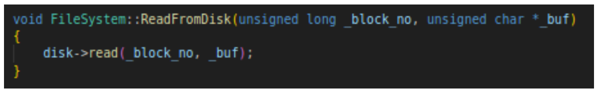


```
void FileSystem::WriteToDisk(unsigned long _block_no, unsigned char *_buf)
{
    disk->write(_block_no, _buf);
}
```

Figure 14: **FileSystem::WriteToDisk()**

(g) **FileSystem::ReadFromDisk()**

For both small and large files, use the read function in SimpleDisk.

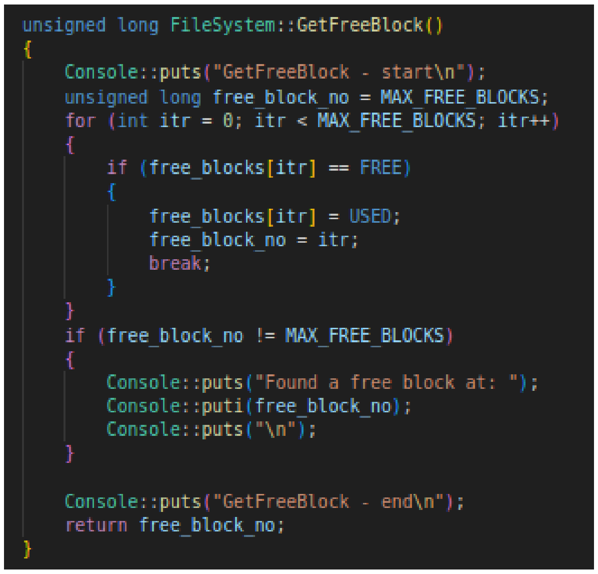


```
void FileSystem::ReadFromDisk(unsigned long _block_no, unsigned char *_buf)
{
    disk->read(_block_no, _buf);
}
```

Figure 15: **FileSystem::ReadFromDisk()**

(h) **unsigned long FileSystem::GetFreeBlock()**

For both small and large files, if there's an available block that is not in use, mark it as used and give it back.

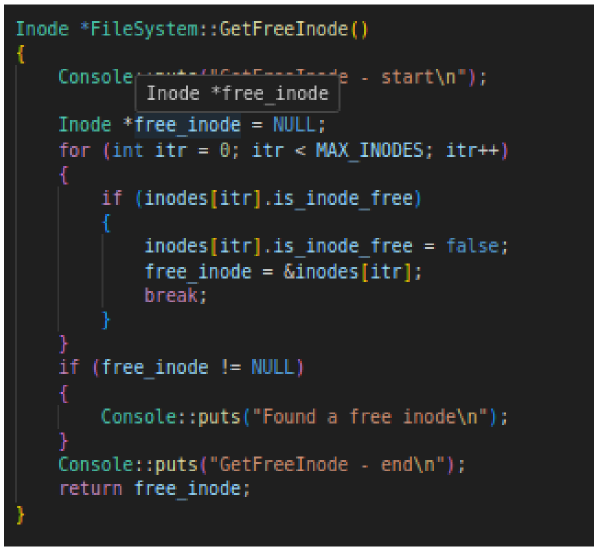


```
unsigned long FileSystem::GetFreeBlock()
{
    Console::puts("GetFreeBlock - start\n");
    unsigned long free_block_no = MAX_FREE_BLOCKS;
    for (int itr = 0; itr < MAX_FREE_BLOCKS; itr++)
    {
        if (free_blocks[itr] == FREE)
        {
            free_blocks[itr] = USED;
            free_block_no = itr;
            break;
        }
    }
    if (free_block_no != MAX_FREE_BLOCKS)
    {
        Console::puts("Found a free block at: ");
        Console::puti(free_block_no);
        Console::puts("\n");
    }

    Console::puts("GetFreeBlock - end\n");
    return free_block_no;
}
```

Figure 16: **unsigned long FileSystem::GetFreeBlock()**

(i) **Inode *FileSystem::GetFreeInode()**

For both small and large files, if there's an unused node available, mark it as used and give it back.

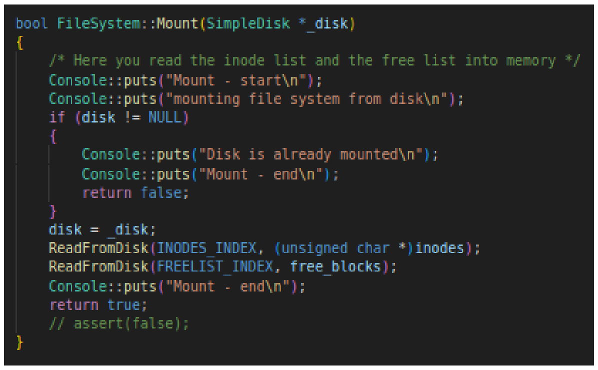


```
Inode *FileSystem::GetFreeInode()
{
    Console::puts("GetFreeInode - start\n");

    Inode *free_inode = NULL;
    for (int itr = 0; itr < MAX_INODES; itr++)
    {
        if (inodes[itr].is_inode_free)
        {
            inodes[itr].is_inode_free = false;
            free_inode = &inodes[itr];
            break;
        }
    }
    if (free_inode != NULL)
    {
        Console::puts("Found a free inode\n");
    }
    Console::puts("GetFreeInode - end\n");
    return free_inode;
}
```

Figure 17: **Inode *FileSystem::GetFreeInode()**

(j) **bool FileSystem::Mount()**

For both small and large files, connect the file system to a disk, and retrieve the inode list and free list from the disk.

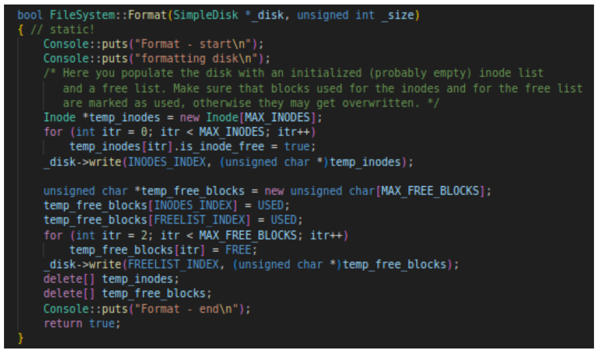


```
bool FileSystem::Mount(SimpleDisk *_disk)
{
    /* Here you read the inode list and the free list into memory */
    Console::puts("Mount - start\n");
    Console::puts("mounting file system from disk\n");
    if (disk != NULL)
    {
        Console::puts("Disk is already mounted\n");
        Console::puts("Mount - end\n");
        return false;
    }
    disk = _disk;
    ReadFromDisk(INODES_INDEX, (unsigned char *)inodes);
    ReadFromDisk(FREELIST_INDEX, free_blocks);
    Console::puts("Mount - end\n");
    return true;
    // assert(false);
}
```

Figure 18: **bool FileSystem::Mount()**

(k) **bool FileSystem::Format()**

For both small and large files, start by setting up the inode list and free list on the disk. Make sure that all inodes and data blocks are marked as available or free. The exception is the first block (for the inode list) and the second block (for the free list), which should be marked as in use.

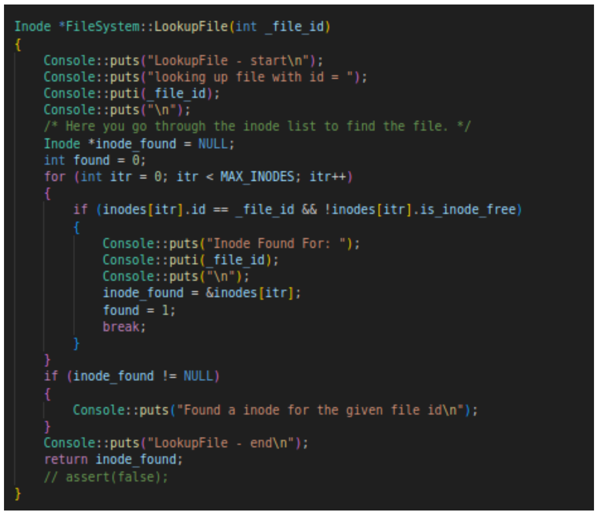


```cpp
bool FileSystem::Format(SimpleDisk *_disk, unsigned int _size)
{ // static!
    Console::puts("Format - start\n");
    Console::puts("formatting disk\n");
    /* Here you populate the disk with an initialized (probably empty) inode list
       and a free list. Make sure that blocks used for the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

    unsigned char *temp_free_blocks = new unsigned char[MAX_FREE_BLOCKS];
    temp_free_blocks[INODES_INDEX] = USED;
    temp_free_blocks[FREELIST_INDEX] = USED;
    for (int itr = 2; itr < MAX_FREE_BLOCKS; itr++)
        temp_free_blocks[itr] = FREE;
    _disk->write(FREELIST_INDEX, (unsigned char *)temp_free_blocks);
    delete[] temp_inodes;
    delete[] temp_free_blocks;
    Console::puts("Format - end\n");
    return true;
}
```

Figure 19: **bool FileSystem::Format()**

(l) **Inode *FileSystem::LookupFile()**

For both small and large files, look through the inode list to locate a specific file.

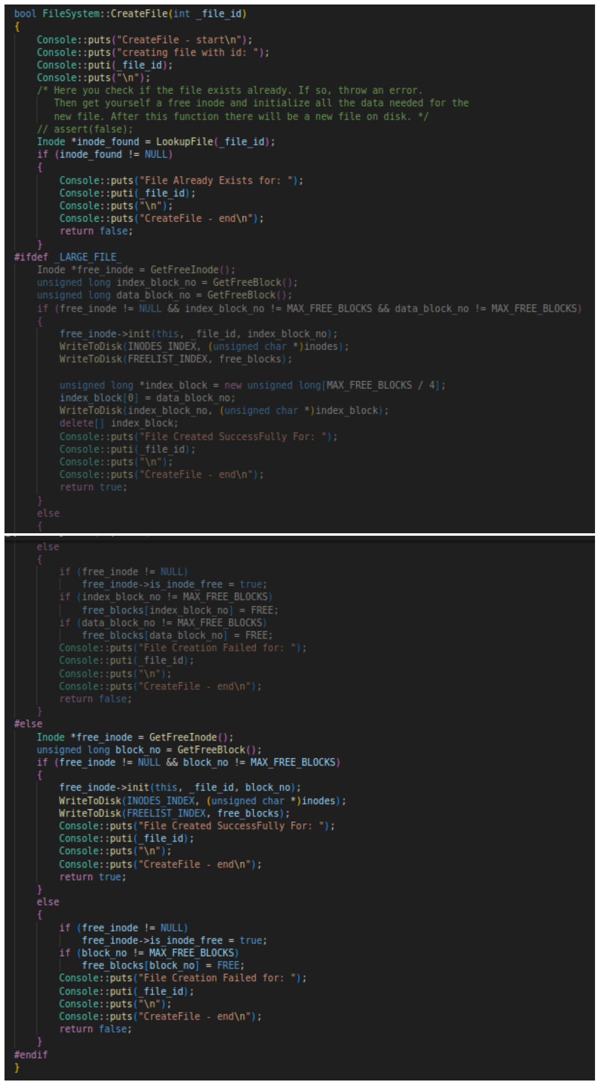


```cpp
Inode *FileSystem::LookupFile(int _file_id)
{
    Console::puts("LookupFile - start\n");
    Console::puts("looking up file with id = ");
    Console::puti(_file_id);
    Console::puts("\n");
    /* Here you go through the inode list to find the file. */
    Inode *inode_found = NULL;
    int found = 0;
    for (int itr = 0; itr < MAX_INODES; itr++)
    {
        if (inodes[itr].id == _file_id && !inodes[itr].is_inode_free)
        {
            Console::puts("Inode Found For: ");
            Console::puti(_file_id);
            Console::puts("\n");
            inode_found = &inodes[itr];
            found = 1;
            break;
        }
    }
    if (inode_found != NULL)
    {
        Console::puts("Found a inode for the given file id\n");
    }
    Console::puts("LookupFile - end\n");
    return inode_found;
    // assert(false);
}
```

Figure 20: **Inode *FileSystem::LookupFile()**

(m) **FileSystem::CreateFile()**

For a small file, if the file name isn't already taken, make a new file. Obtain a free inode and a free block, then set up the inode. Finally, update the inode list and free list on the disk. For a large file, if the file name isn't in use, create a new file. Get a free inode and two free blocks—one for the index block and another for the data block. Initialize the inode and update the inode list, free list, and index block on the disk.



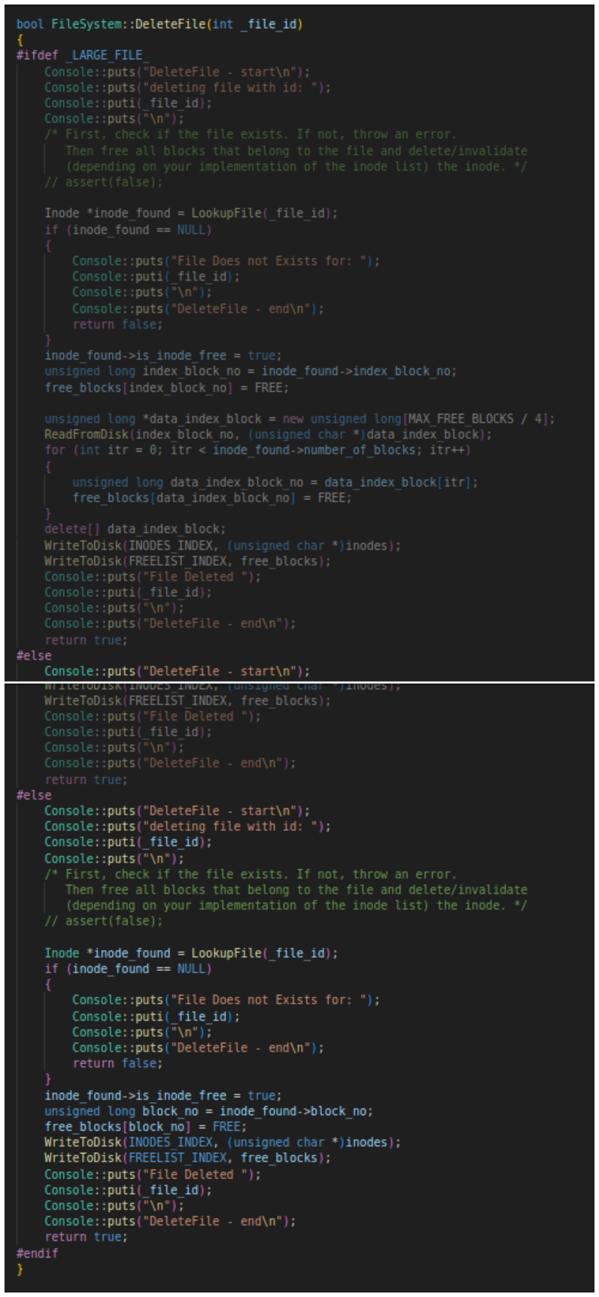
```cpp
bool FileSystem::CreateFile(int _file_id)
{
    Console::puts("CreateFile - start\n");
    Console::puts("creating file with id: ");
    Console::puti(_file_id);
    Console::puts("\n");
    /* Here you check if the file exists already. If so, throw an error.
       Then get yourself a free inode and initialize all the data needed for the
       new file. After this function there will be a new file on disk. */
    // assert(false);
    Inode *inode_found = LookupFile(_file_id);
    if (inode_found != NULL)
    {
        Console::puts("File Already Exists for: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("CreateFile - end\n");
        return false;
    }
#ifdef _LARGE_FILE_
    Inode *free_inode = GetFreeInode();
    unsigned long index_block_no = GetFreeBlock();
    unsigned long data_block_no = GetFreeBlock();
    if (free_inode != NULL && index_block_no != MAX_FREE_BLOCKS && data_block_no != MAX_FREE_BLOCKS)
    {
        free_inode->init(this, _file_id, index_block_no);
        WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
        WriteToDisk(FREELIST_INDEX, free_blocks);

        unsigned long *index_block = new unsigned long[MAX_FREE_BLOCKS / 4];
        index_block[0] = data_block_no;
        WriteToDisk(index_block_no, (unsigned char *)index_block);
        delete[] index_block;
        Console::puts("File Created SuccessFully For: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("CreateFile - end\n");
        return true;
    }
    else
    {
```

```cpp
    else
    {
        if (free_inode != NULL)
            free_inode->is_inode_free = true;
        if (index_block_no != MAX_FREE_BLOCKS)
            free_blocks[index_block_no] = FREE;
        if (data_block_no != MAX_FREE_BLOCKS)
            free_blocks[data_block_no] = FREE;
        Console::puts("File Creation Failed for: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("CreateFile - end\n");
        return false;
    }
#else
    Inode *free_inode = GetFreeInode();
    unsigned long block_no = GetFreeBlock();
    if (free_inode != NULL && block_no != MAX_FREE_BLOCKS)
    {
        free_inode->init(this, _file_id, block_no);
        WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
        WriteToDisk(FREELIST_INDEX, free_blocks);
        Console::puts("File Created SuccessFully For: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("CreateFile - end\n");
        return true;
    }
    else
    {
        if (free_inode != NULL)
            free_inode->is_inode_free = true;
        if (block_no != MAX_FREE_BLOCKS)
            free_blocks[block_no] = FREE;
        Console::puts("File Creation Failed for: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("CreateFile - end\n");
        return false;
    }
#endif
}
```

Figure 21: **FileSystem::CreateFile()**

(n) **FileSystem::DeleteFile()**

For a small file, if the file name is already taken, remove the file. Mark both the inode and the data block as available or free, and then update the inode list and free list on the disk. For a large file, if the file name is in use, delete the file. Mark the inode, index block, and all data blocks as free, and then update the inode list and free list on the disk.



```
bool FileSystem::DeleteFile(int _file_id)
{
#ifdef _LARGE_FILE_
    Console::puts("DeleteFile - start\n");
    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. */
    // assert(false);

    Inode *inode_found = LookupFile(_file_id);
    if (inode_found == NULL)
    {
        Console::puts("File Does not Exists for: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("DeleteFile - end\n");
        return false;
    }
    inode_found->is_inode_free = true;
    unsigned long index_block_no = inode_found->index_block_no;
    free_blocks[index_block_no] = FREE;

    unsigned long *data_index_block = new unsigned long[MAX_FREE_BLOCKS / 4];
    ReadFromDisk(index_block_no, (unsigned char *)data_index_block);
    for (int itr = 0; itr < inode_found->number_of_blocks; itr++)
    {
        unsigned long data_index_block_no = data_index_block[itr];
        free_blocks[data_index_block_no] = FREE;
    }
    delete[] data_index_block;
    WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
    WriteToDisk(FREELIST_INDEX, free_blocks);
    Console::puts("File Deleted ");
    Console::puti(_file_id);
    Console::puts("\n");
    Console::puts("DeleteFile - end\n");
    return true;
#else
    Console::puts("DeleteFile - start\n");
```

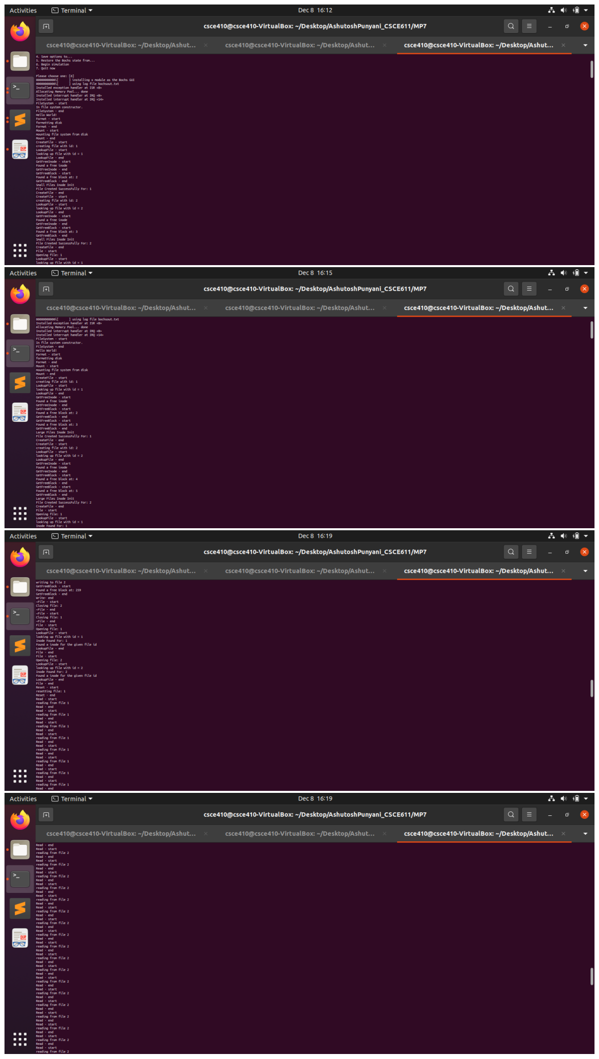
```
    WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
    WriteToDisk(FREELIST_INDEX, free_blocks);
    Console::puts("File Deleted ");
    Console::puti(_file_id);
    Console::puts("\n");
    Console::puts("DeleteFile - end\n");
    return true;
#else
```

```
    Console::puts("DeleteFile - start\n");
    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. */
    // assert(false);

    Inode *inode_found = LookupFile(_file_id);
    if (inode_found == NULL)
    {
        Console::puts("File Does not Exists for: ");
        Console::puti(_file_id);
        Console::puts("\n");
        Console::puts("DeleteFile - end\n");
        return false;
    }
    inode_found->is_inode_free = true;
    unsigned long block_no = inode_found->block_no;
    free_blocks[block_no] = FREE;
    WriteToDisk(INODES_INDEX, (unsigned char *)inodes);
    WriteToDisk(FREELIST_INDEX, free_blocks);
    Console::puts("File Deleted ");
    Console::puti(_file_id);
    Console::puts("\n");
    Console::puts("DeleteFile - end\n");
    return true;
#endif
}
```

Figure 22: **FileSystem::DeleteFile()**

Testing

During the development of the code, I wrote several `Console:puts()` and `Console:putui()` statements to pinpoint where my code was breaking and to understand if the logic was incorrect or not performing as expected. I removed some Console statements. To check the bonus option, I modified the exercisefilesystem function in kernel.c. Everything else in the process remains the same as the original function, except now 64KB of data will be written into the file. The choice between the original and modified functions is determined by using _LARGE_FILE_ in all five files that I have edited.



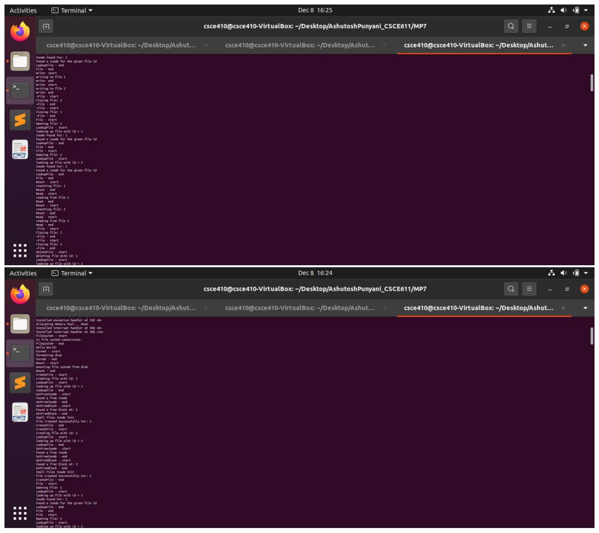
```
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
FileSystem - start
In file system constructor.
FileSystem - end
Hello World!
Format - start
formatting disk
Format - end
Mount - start
mounting file system from disk
Mount - end
CreateFile - start
creating file with id: 1
LookupFile - start
looking up file with id = 1
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 2
GetFreeBlock - end
Small Files Inode Init
File Created SuccessFully For: 1
CreateFile - end
CreateFile - start
creating file with id: 2
LookupFile - start
looking up file with id = 2
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 3
GetFreeBlock - end
Small Files Inode Init
File Created SuccessFully For: 2
CreateFile - end
File - start
Opening file: 1
LookupFile - start
looking up file with id = 1
```

```
00000000000i[      ] using log file bochsout.txt
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
FileSystem - start
In file system constructor.
FileSystem - end
Hello World!
Format - start
formatting disk
Format - end
Mount - start
mounting file system from disk
Mount - end
CreateFile - start
creating file with id: 1
LookupFile - start
looking up file with id = 1
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 2
GetFreeBlock - end
GetFreeBlock - start
Found a free block at: 3
GetFreeBlock - end
Large Files Inode Init
File Created SuccessFully For: 1
CreateFile - end
CreateFile - start
creating file with id: 2
LookupFile - start
looking up file with id = 2
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 4
GetFreeBlock - end
GetFreeBlock - start
Found a free block at: 5
GetFreeBlock - end
Large Files Inode Init
File Created SuccessFully For: 2
CreateFile - end
File - start
Opening file: 1
LookupFile - start
looking up file with id = 1
Inode Found For: 1
```

```
writing to file 2
GetFreeBlock - start
Found a free block at: 259
GetFreeBlock - end
Write- end
~File - start
Closing file: 2
~File - end
~File - start
Closing file: 1
~File - end
File - start
Opening file: 1
LookupFile - start
looking up file with id = 1
Inode Found For: 1
Found a inode for the given file id
LookupFile - end
File - end
File - start
Opening file: 2
LookupFile - start
looking up file with id = 2
Inode Found For: 2
Found a inode for the given file id
LookupFile - end
File - end
Reset - start
resetting file: 1
Reset - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
Read - start
reading from file 1
Read - end
```

```
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
Read - end
Read - start
reading from file 2
```

Figure 23: **Testing (with _LARGE_FILE_ uncommented)**



```
Inode Found For: 2
Found a inode for the given file id
LookupFile - end
File - end
Write- start
writing to file 1
Write- end
Write- start
writing to file 2
Write- end
~File - start
Closing file: 2
~File - end
~File - start
Closing file: 1
~File - end
File - start
Opening file: 1
LookupFile - start
looking up file with id = 1
Inode Found For: 1
Found a inode for the given file id
LookupFile - end
File - end
File - start
Opening file: 2
LookupFile - start
looking up file with id = 2
Inode Found For: 2
Found a inode for the given file id
LookupFile - end
File - end
Reset - start
resetting file: 1
Reset - end
Read - start
reading from file 1
Read - end
Reset - start
resetting file: 2
Reset - end
Read - start
reading from file 2
Read - end
~File - start
Closing file: 2
~File - end
~File - start
Closing file: 1
~File - end
DeleteFile - start
deleting file with id: 1
LookupFile - start
looking up file with id = 1
```

```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <14>
FileSystem - start
In file system constructor.
FileSystem - end
Hello World!
Format - start
formatting disk
Format - end
Mount - start
mounting file system from disk
Mount - end
CreateFile - start
creating file with id: 1
LookupFile - start
looking up file with id = 1
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 2
GetFreeBlock - end
Small Files Inode Init
File Created SuccessFully For: 1
CreateFile - end
CreateFile - start
creating file with id: 2
LookupFile - start
looking up file with id = 2
LookupFile - end
GetFreeInode - start
Found a free inode
GetFreeInode - end
GetFreeBlock - start
Found a free block at: 3
GetFreeBlock - end
Small Files Inode Init
File Created SuccessFully For: 2
CreateFile - end
File - start
Opening file: 1
LookupFile - start
looking up file with id = 1
Inode Found For: 1
Found a inode for the given file id
LookupFile - end
File - end
File - start
Opening file: 2
LookupFile - start
looking up file with id = 2
```

Figure 24: **Testing (with _LARGE_FILE_ commented)**