Kwangkyu Hwang

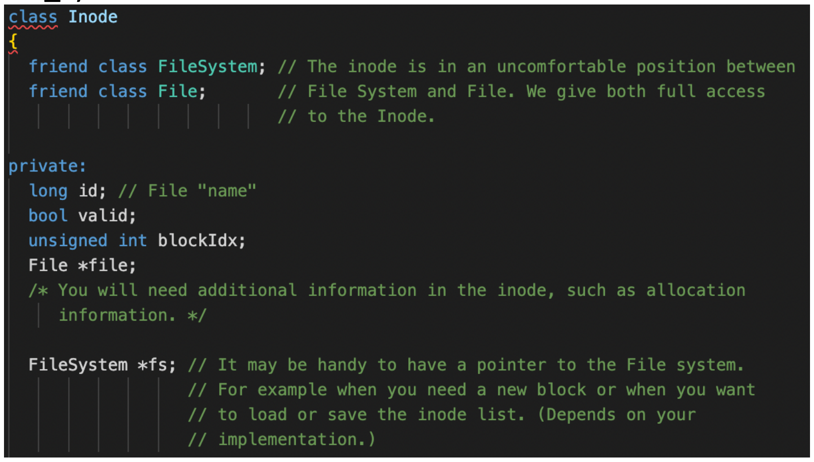
MP7 design

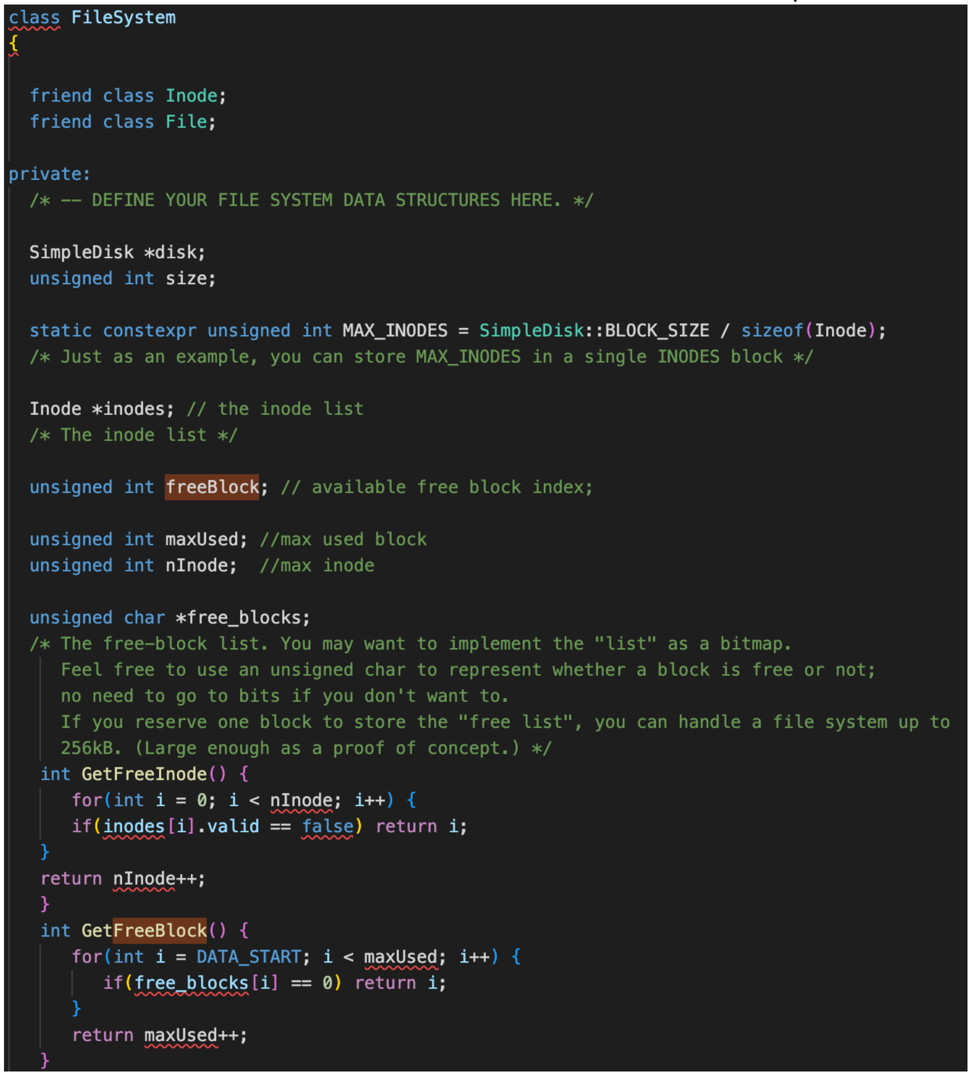
In kernel.C, FileSystem is constructed, 128KB file system is Formatted and Mounted. After that to check FileSystem, two files are crated and two strings are written to files(store to block_cache in file). When file object destruct when leave scope, write block_cache data to disk. To verify string is well written to file, open files again and read strings from files and compare with original string. After finishing comparison, delete files.

Main modification code of MP7 file.H, file.C, file_system.H, file_system.C. I also added additional consideration for <OPTION 1>

Kwangkyu Hwang

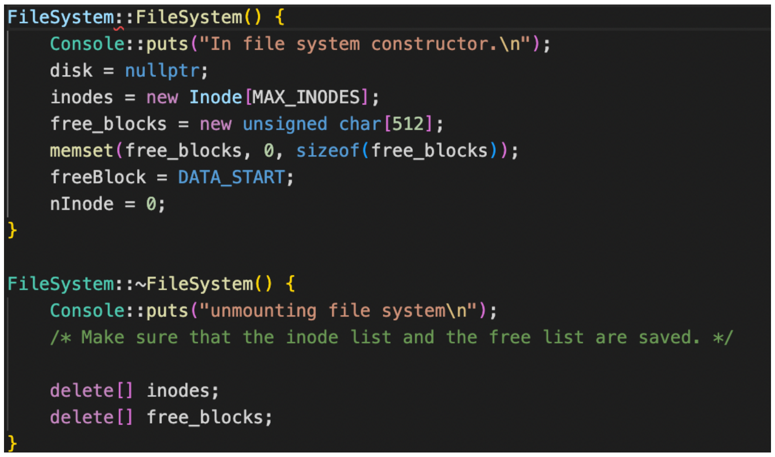
FileSystem
file_syste.H


```
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"
  bool valid;
  unsigned int blockIdx;
  File *file;
  /* 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.)
```
In Inode class, I added valid, blockIdx, file. Valid equals true means Inode is in use, and if it is false, it is not in use. blockIdx means file's location in data block which is start from 2 since 0 and 1 is reserved for inode list and free list. Also I added File pointer in Inode.


```
class FileSystem
{

  friend class Inode;
  friend class File;

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

  SimpleDisk *disk;
  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 int freeBlock; // available free block index;

  unsigned int maxUsed; //max used block
  unsigned int nInode;  //max inode

  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.) */
   int GetFreeInode() {
      for(int i = 0; i < nInode; i++) {
      if(inodes[i].valid == false) return i;
   }
   return nInode++;
   }
   int GetFreeBlock() {
      for(int i = DATA_START; i < maxUsed; i++) {
         if(free_blocks[i] == 0) return i;
      }
      return maxUsed++;
   }
```

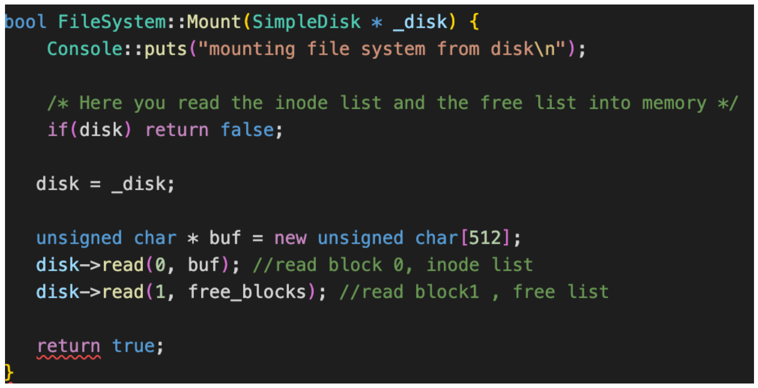
Kwangkyu Hwang

File_system.C
In the FileSystem class, disk, size, freeBlock, maxUsed, nInode are added and GetFreeInode, GetFreeBlock methods are implemented. freeBlock is used for available free block index. maxUsed variable means max used block index in FileSystem and nInode means max inode index in FileSystem. Based on this information and checking valid variables of inode in inode_list and used bitmap information of free_blocks, we can get free inode and free block index. Also increased max values since it always called when create file.



```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");
    disk = nullptr;
    inodes = new Inode[MAX_INODES];
    free_blocks = new unsigned char[512];
    memset(free_blocks, 0, sizeof(free_blocks));
    freeBlock = DATA_START;
    nInode = 0;
}

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

    delete[] inodes;
    delete[] free_blocks;
}
```

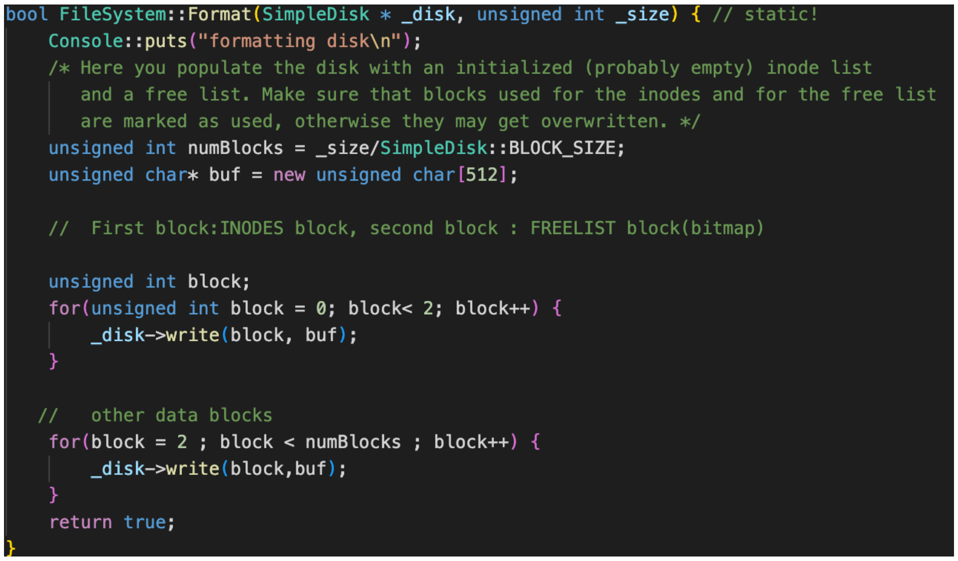
Filesystem constructor initialize inode_list and free_blocks(free_list). freeBlock is initialized to 2 since block0 and block1 is in use. In destructor deleted inode_list and free_blocks.



```
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) return false;

   disk = _disk;

   unsigned char * buf = new unsigned char[512];
   disk->read(0, buf); //read block 0, inode list
   disk->read(1, free_blocks); //read block1 , free list

   return true;
}
```

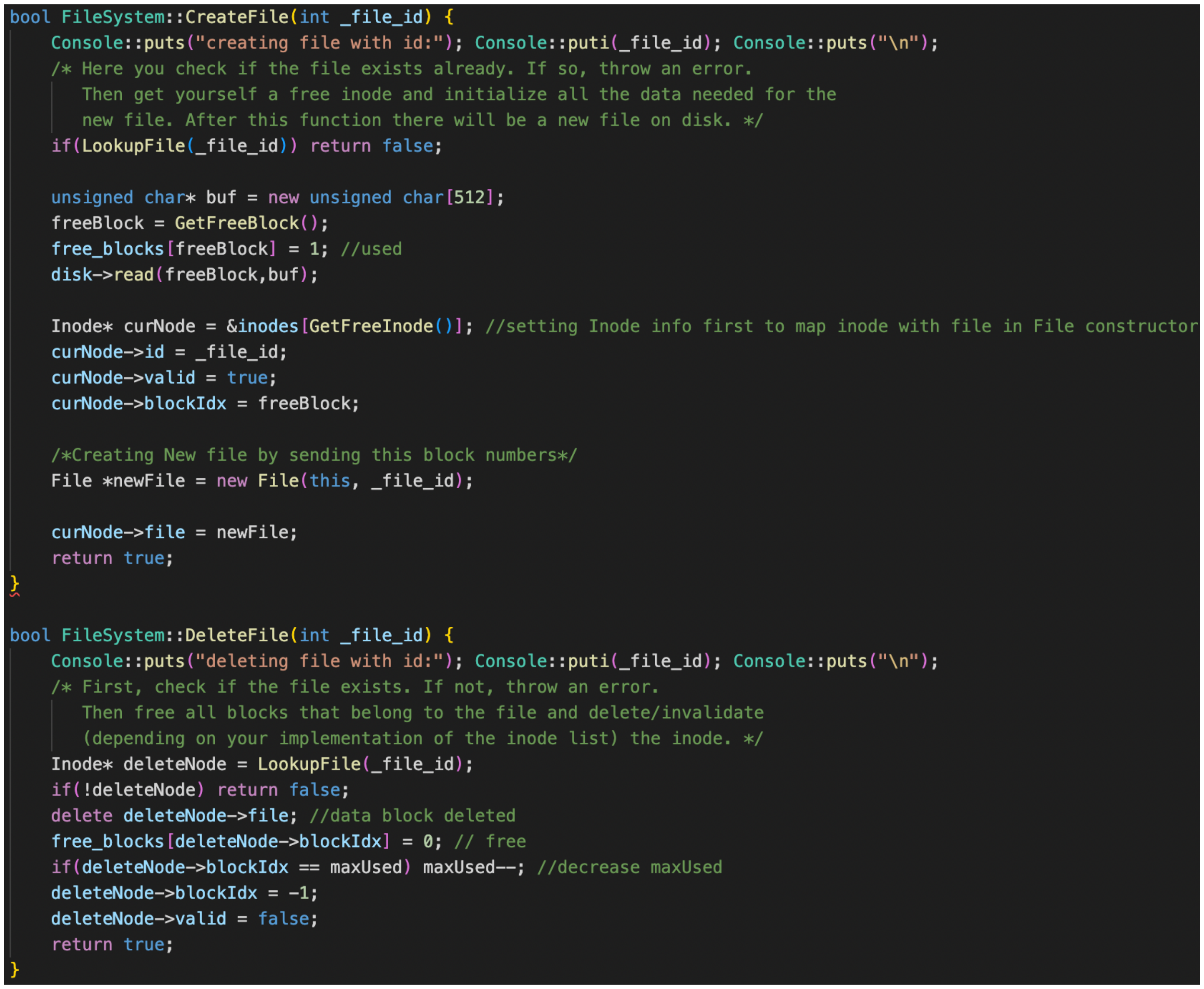
In filesystem, disk should mount once, thus check if disk is set. If not set disk and read block 0 and block 1 from disk to get information of file system.

Kwangkyu Hwang



```
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. */
    unsigned int numBlocks = _size/SimpleDisk::BLOCK_SIZE;
    unsigned char* buf = new unsigned char[512];

    //  First block:INODES block, second block : FREELIST block(bitmap)

    unsigned int block;
    for(unsigned int block = 0; block< 2; block++) {
        _disk->write(block, buf);
    }

   //   other data blocks
    for(block = 2 ; block < numBlocks ; block++) {
        _disk->write(block,buf);
    }
    return true;
}
```

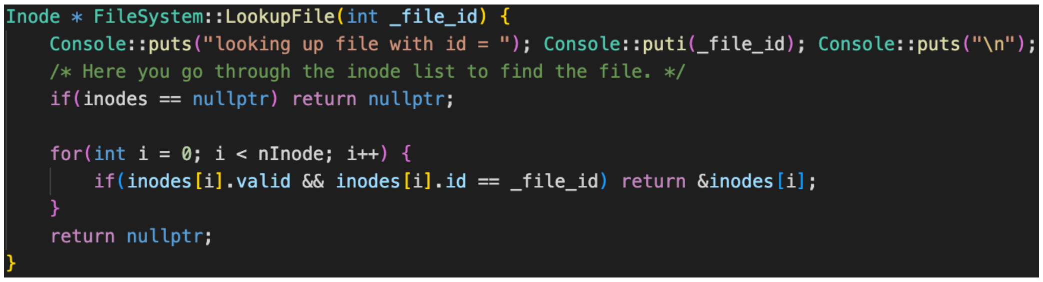
In format method, how many data blocks need to make '_size' size filesystem is calculated. In disk amout of that blocks are reset.



```
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)) return false;

    unsigned char* buf = new unsigned char[512];
    freeBlock = GetFreeBlock();
    free_blocks[freeBlock] = 1; //used
    disk->read(freeBlock,buf);

    Inode* curNode = &inodes[GetFreeInode()]; //setting Inode info first to map inode with file in File constructor
    curNode->id = _file_id;
    curNode->valid = true;
    curNode->blockIdx = freeBlock;

    /*Creating New file by sending this block numbers*/
    File *newFile = new File(this, _file_id);

    curNode->file = newFile;
    return true;
}

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* deleteNode = LookupFile(_file_id);
    if(!deleteNode) return false;
    delete deleteNode->file; //data block deleted
    free_blocks[deleteNode->blockIdx] = 0; // free
    if(deleteNode->blockIdx == maxUsed) maxUsed--; //decrease maxUsed
    deleteNode->blockIdx = -1;
    deleteNode->valid = false;
    return true;
}
```

Kwangkyu Hwang

CreateFile and DeleteFile do opposite functions. When create file, first check file already exist. If not, get Free block index and inode from predefined method(getFreeBlock and getFreeInode). Set those information as used and valid and create new File and set inode's file pointer refer to newly created file.
 DeleteFile find inode using LookupFile method and delete file using Inode and update information of inode_list and free_list.

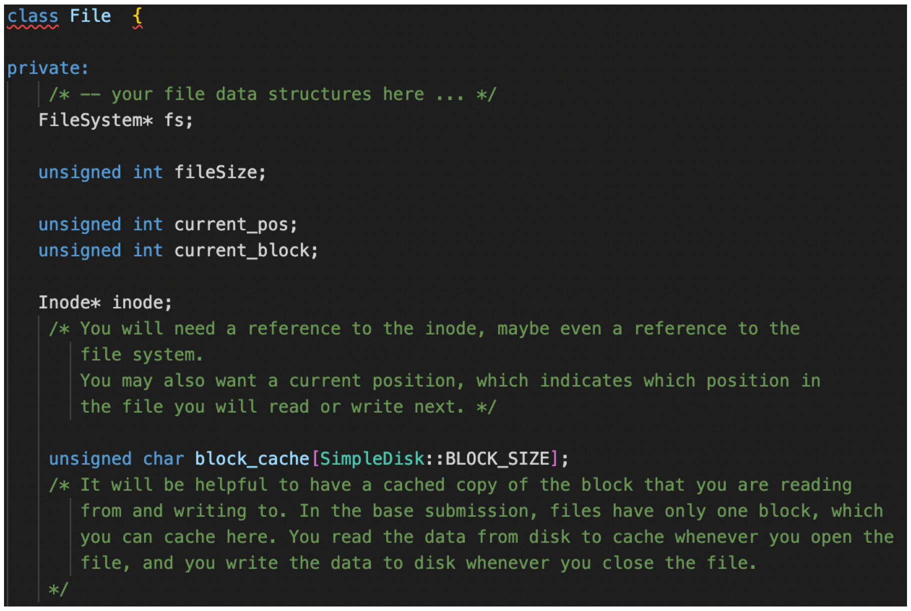


```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::puti(_file_id); Console::puts("\n");
    /* Here you go through the inode list to find the file. */
    if(inodes == nullptr) return nullptr;

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

LookupFile find inode from inode_list using _file_id. If find inode return inode address and if not return nullptr

File

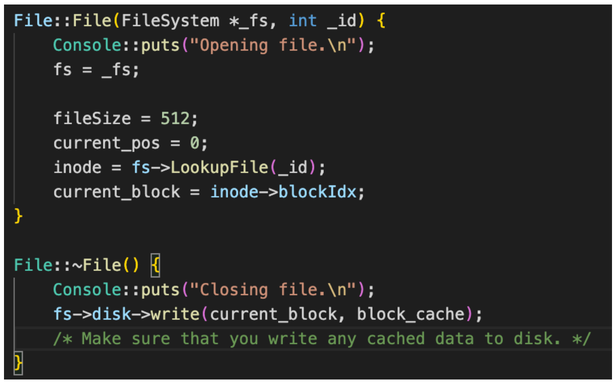
File.H



```
class File  {

private:
    /* -- your file data structures here ... */
   FileSystem* fs;

   unsigned int fileSize;

   unsigned int current_pos;
   unsigned int current_block;

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

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

File contains FileSystem fs, file size and current_block and current_pos, inode, block_cache. Since in our project, File size is restricted to only one block, we can set that block index as current_block and current_pos is indicate that the data is written from zero to current position in block which can range from 0-511. Also inode is the inode that indicate current file and block_cache stores read data from disk.

File.C

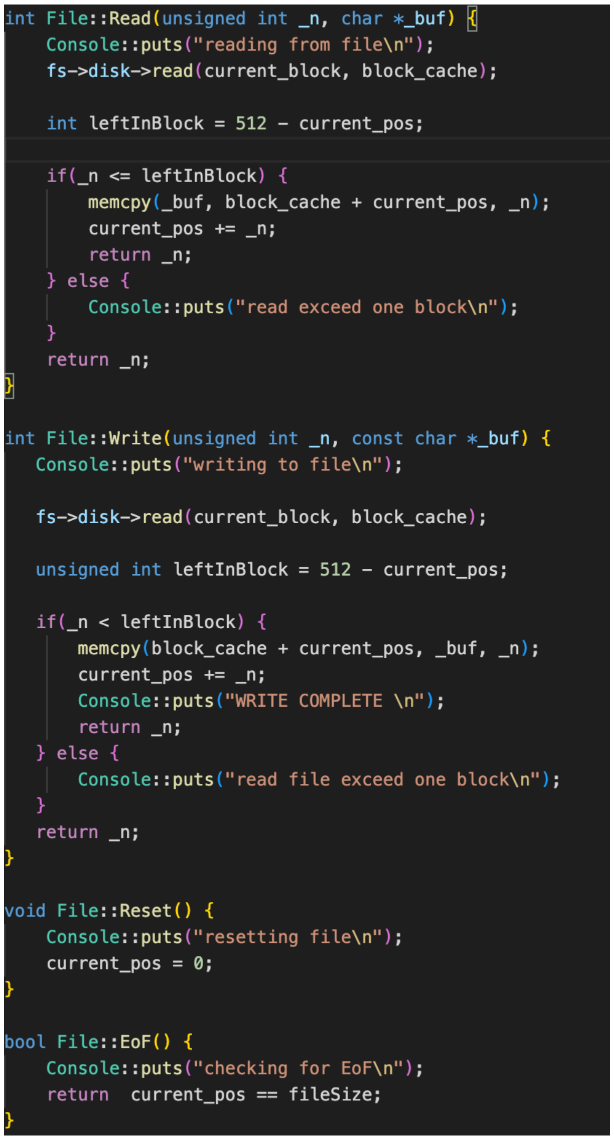


```
File::File(FileSystem *_fs, int _id) {
    Console::puts("Opening file.\n");
    fs = _fs;

    fileSize = 512;
    current_pos = 0;
    inode = fs->LookupFile(_id);
    current_block = inode->blockIdx;
}

File::~File() {
    Console::puts("Closing file.\n");
    fs->disk->write(current_block, block_cache);
    /* Make sure that you write any cached data to disk. */
}
```

In constructor, variables are initialized and file system is set. Inode is get from file system and current_block can get from inode.

In destructor, cached data should written to the disk. Thus using filesystem, write data in block_cache to the disk in current_block index block.

Kwangkyu Hwang



```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    fs->disk->read(current_block, block_cache);

    int leftInBlock = 512 - current_pos;

    if(_n <= leftInBlock) {
        memcpy(_buf, block_cache + current_pos, _n);
        current_pos += _n;
        return _n;
    } else {
        Console::puts("read exceed one block\n");
    }
    return _n;
}

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

   fs->disk->read(current_block, block_cache);

   unsigned int leftInBlock = 512 - current_pos;

   if(_n < leftInBlock) {
       memcpy(block_cache + current_pos, _buf, _n);
       current_pos += _n;
       Console::puts("WRITE COMPLETE \n");
       return _n;
   } else {
       Console::puts("read file exceed one block\n");
   }
   return _n;
}

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

bool File::EoF() {
    Console::puts("checking for EoF\n");
    return  current_pos == fileSize;
}
```

Read method read _n data from disk and copy it to _buf. Thus, check whether current_pos + _n is not exceed blocksize(512).

Write method also read data from disk and add data from _buf to block_cache. And update current position. As comment of original code says, writing to disk is executed when file is closed(destructor)

Reset just make current_position to 0

EoF means data is already full in this file block.

Kwangkyu Hwang

Additional consideration for <OPTION 1: DESIGN of an extension to the basic file system to allow for files that are up to 64kB long.>

Our basic design just allowed 1 data block for one file (512Byte). To make 64KB file, maximum 128 data blocks are used for one file. This data blocks can be implemented by variable size array or list. Thus, in File class, block_cache also be array of blocks not just a block. Also Inode should be modified to have array of indices that allocated to matching files
And Write and Read methods in File should be modified. Since we only handled one block for file, we just store position in a block. But now we need to store the current position in last block and number of blocks that have been written. Let's assume that the first block is filled with data up to 500 (assuming 1 index), and then 20 data is about to be written. In this case, since the size of one data block is exceeded, 12 data should be written to that block, and data of 8 should be written to a new block. In this case, a new block will allocated from the FileSystem and data of 8 is written to the block. Even in the case of Read, logic that traverses the blocks must be added to read from the first block to the end.