MP6: Primitive Disk Device Driver

Kai-Chih Huang
UIN: 333000021
CSCE611: Operating System

Assigned Tasks

Main Part: Finished
Bonus Option 1: Finished
Bonus Option 2: Finished

System Design

The goal of this machine problem is to implement a simple file system. I this file system, I used 1 disk block to store the inodes, and 1 disk block to keep track of the free blocks. For the bonus option 1, I designed a 2-level block id structure to store files that requires multiple blocks. To be more specific, I change the block_id attribute in the Inode data structure to store the ids of the blocks used by the file, instead of storing a single block.

Code Description

I implemented file_system.H/C and file.H/C. In addition, I changed some codes in kernel.C for testing.

To compile code, simply run following command lines under MP6_Sources directory:

```
$ make clean    // clean the old compile files before we compile

$ make          // compile files

$ ./copykernel.sh (if permission denied, try chmod u+x ./copykernel.sh then do it again)    //copy kernel

$ bochs -f bochsrc.bxrc     // run bochs
```

I will walk **through the** functions/methods defined in above files as follows.

file_system.H

I defined Inode class and file_system class in this file. Inode class contains 4 attributes, which is id(file name), block_id(list of block ids that store the file for bonus option task), size(file size in bytes) and a pointer to the

FileSystem class.

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

public:
  long id; // File "name"
  int block_id; // Block where the file is stored.
  int size; // File size in bytes.

  /* 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. */

};
```

As for the FileSystem class, it has the disk size attribute, the inode list, a freeblock list, and a pointer to the SimpleDisk class. I added a GetFreeBlock function to simply find and return the id of a free block.

```
class FileSystem
{

  friend class Inode;

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

  unsigned int size;

  static constexpr unsigned int MAX_INODES = SimpleDisk::BLOCK_SIZE / sizeof(Inode);
  /* Just as an example, you can store MAX_INODES in a single INODES block */

  Inode *inodes; // the inode list
  /* The inode list */

  unsigned char *free_blocks;
  /* The free-block list. You may want to implement the "list" as a bitmap.
     Feel free to use an unsigned char to represent whether a block is free or not;
     no need to go to bits if you don't want to.
     If you reserve one block to store the "free list", you can handle a file system up to
     256kB. (Large enough as a proof of concept.) */

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

public:
  SimpleDisk *disk;
```

```
public:
  SimpleDisk *disk;

  FileSystem();
  /* Just initializes local data structures. Does not connect to disk yet. */

  ~FileSystem();
  /* Unmount file system if it has been mounted. */

  bool Mount(SimpleDisk *_disk);
  /* Associates this file system with a disk. Limit to at most one file system per disk.
     Returns true if operation successful (i.e. there is indeed a file system on the disk.) */

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

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

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

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

  int GetFreeBlock();
  /* Find a free block in the disk and return the block id. If no block found, return -1. */
};
#endif
```

file_system.C

FileSystem::FileSystem()
This is the constructor that simply initialize the attributes, nothing special

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

FileSystem::~FileSystem()
The destructor will first write back the inode list and free block list to disk block 0 and disk block 1

correspondingly, then free the memory.

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

    // Write the inode list to block 0
    disk->write(0, (unsigned char *) inodes);
    // Write the free list to block 1
    disk->write(1, free_blocks);

    delete inodes;
    delete free_blocks;
}
```

bool FileSystem::Mount(SimpleDisk * _disk)
This function connect the file system to the disk. It will read the first disk block to inode list, and read the second block as the free block list.

```
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 */

    disk = _disk;
    size = _disk->size();

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

    free_blocks = new unsigned char[SimpleDisk::BLOCK_SIZE];
    disk->read(1, free_blocks);

    delete temp_inode_block;
    return true;
}
```

bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size)
It set the inodes list with size MAX_INODES to 0 and write to block 0, then m ark the 1st and 2nd position as used and write to the disk at block 1.

```
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. */

    // Set the inodes list with size MAX_INODES to 0 and write to block 0
    Inode * empty_inode_block = new Inode[MAX_INODES];
    memset(empty_inode_block, 0, MAX_INODES);
    _disk->write(0, (unsigned char *) empty_inode_block);

    // Mark the 1st and 2nd position as used and write to the disk at block 1
    unsigned char empty_free_block[SimpleDisk::BLOCK_SIZE];
    memset(empty_free_block, 0, SimpleDisk::BLOCK_SIZE);
    empty_free_block[0] = 1;
    empty_free_block[1] = 1;
    _disk->write(1, empty_free_block);

    return true;
}
```

bool FileSystem::CreateFile(int _file_id)
Before creating a new file, we first check if the file name is already existed, and if there is free block. Then, we initialize a new inode for the file.

```
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. */
    int free_block_id = GetFreeBlock();

    if(LookupFile(_file_id) != NULL || free_block_id == -1) {
        Console::puts("file already exists or no free block found\n");
        return false;
    }
    // Find a free inode and initialize it
    for(int i = 0 ; i < MAX_INODES; i++) {
        if(inodes[i].id == 0) {
            inodes[i].id = _file_id;
            inodes[i].block_id = free_block_id;
            inodes[i].fs = this;
            inodes[i].size = 0;

            // Mark the block as used in the free list
            free_blocks[inodes[i].block_id] = 1;
            return true;
        }
    }
    // If no free inode is found, return false
    return false;
}
```

bool FileSystem::DeleteFile(int _file_id)
Same as creating a file, we also need to check if the file exists before we delete the file. Then we free all the blocks in the block_id list and the block of block list itself. Finally we invalidate inode and return

```
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 * file_inode = LookupFile(_file_id);
    if(file_inode == NULL) {
        return false;
    }

    // First we need to free all the blocks that belong to the file. All the blocks ids are stored in inod
    unsigned char block_ids[SimpleDisk::BLOCK_SIZE];
    disk->read(file_inode->block_id, block_ids);
    for(int i = 0 ; i < SimpleDisk::BLOCK_SIZE; i++) {
        if(block_ids[i] != 0) {
            free_blocks[block_ids[i]] = 0;
        }
    }

    // Then we can free the block_id of the inode itself
    free_blocks[file_inode->block_id] = 0;

    // Finally we can invalidate the inode
    file_inode->id = 0;
    file_inode->block_id = 0;
    file_inode->fs = NULL;
    file_inode->size = 0;

    return true;
}
```

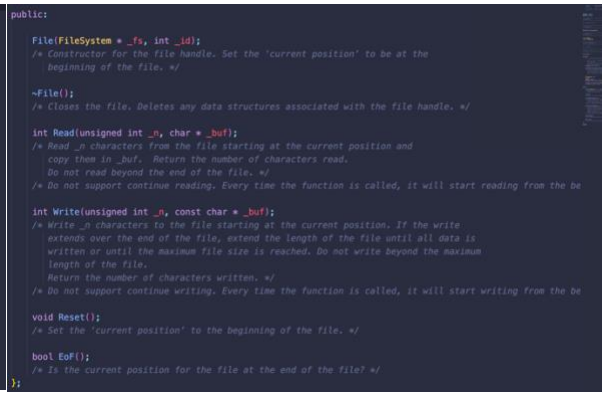
int FileSystem::GetFreeBlock()
The function iterates through the free_block list and find a free block, wipe it clean then return the id of that block. If there is no free block, it will return -1.

```
int FileSystem::GetFreeBlock() {
    Console::puts("getting free block\n");
    /* Here you go through the free list to find a free block. */

    for(int i = 0 ; i < SimpleDisk::BLOCK_SIZE; i++) {
        if(free_blocks[i] == 0) {
            // Wipe the block
            unsigned char empty_block[SimpleDisk::BLOCK_SIZE];
            memset(empty_block, 0, SimpleDisk::BLOCK_SIZE);
            disk->write(i, empty_block);

            // Mark the block as used in the free list
            free_blocks[i] = 1;
            return i;
        }
    }
    // If no free block is found, return -1
    Console::puts("no free block found!!\n");
    return -1;
}
```

file.H

I added some attributes to the File class. The curr_pos is the current position of the file indicates which position in the file that the will be read or written next. In addition, I added the block_ids to store the content of inode.block_id, which a list of blocks used by the file. Finally, I set the MAX_FILE_SIZE to indicate the maximum file size allowed, which is 64KB.

```
class File {

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

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

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

    FileSystem * fs;
    int id;
    Inode * inode;
    unsigned char block_ids[SimpleDisk::BLOCK_SIZE];

    unsigned int MAX_FILE_SIZE = SimpleDisk::BLOCK_SIZE * 128;
    /* MAX_FILE_SIZE = 64KB = 128 blocks */
```

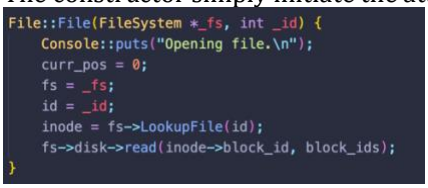


```
public:

    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. */
    /* Do not support continue reading. Every time the function is called, it will start reading from the be

    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. */
    /* Do not support continue writing. Every time the function is called, it will start writing from the be

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

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

file.C

File::File(FileSystem *_fs, int _id)

The constructor simply initiate the attributes in the class.



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

File::~File()

The destructor simply frees the memory allocated when creating the file instance. Note that we don't write back any cached data since we already write back everything needed when reading or writing.

```
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. */

    delete block_cache;
    delete block_ids;
}
```

int File::Read(unsigned int _n, char *_buf)

It will iterate given times, each time read 1 byte to the buffer. Every time when curr_pos % SimpleDisk::BLOCK_SIZE == 0, that means it is at the beginning of a disk block, so we load that block to the block_buffer, then read the content in that buffer.

```
int File::Read(unsigned int _n, char *_buf) {
    Console::puts("reading from file\n");
    for(int i = 0; i < _n; i++) {
        if(EoF()) {
            Console::puts("reached end of file\n");
            Reset();
            return i;
        }

        if(curr_pos % SimpleDisk::BLOCK_SIZE == 0) {
            // Read the next block from disk to block_cache
            fs->disk->read(block_ids[(curr_pos / SimpleDisk::BLOCK_SIZE)], block_cache);
        }

        _buf[i] = block_cache[curr_pos % SimpleDisk::BLOCK_SIZE];
        curr_pos++;
    }
    Reset();
    return _n;
}
```

int File::Write(unsigned int _n, const char *_buf)
It will iterate given times, each time write 1 byte to the buffer. Every time when curr_pos % SimpleDisk::BLOCK_SIZE == 0, that means the buffer is full. Then we write back the buffer, find a new free block and wipe out the buffer and start writing again.

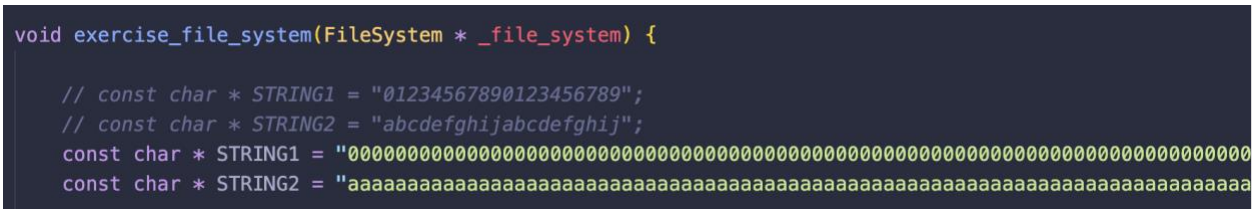
```
int File::Write(unsigned int _n, const char *_buf) {
    Console::puts("writing to file\n");
    for(int i = 0; i < _n; i++) {
        if(curr_pos == MAX_FILE_SIZE) {
            Console::puts("file exceeds limit: 64KB\n");
            return i;
        }

        if(curr_pos % SimpleDisk::BLOCK_SIZE == 0) { // The current block is full or 0
            if (curr_pos != 0) {
                // Write the current fulled block_cache to disk
                fs->disk->write(block_ids[(curr_pos / SimpleDisk::BLOCK_SIZE) - 1], block_cache);
            }
            // We need to prepare a new block
            block_ids[curr_pos / SimpleDisk::BLOCK_SIZE] = fs->GetFreeBlock(); // Here we ignore the possibl
            // Clear the block_cache
            for(int j = 0; j < SimpleDisk::BLOCK_SIZE; j++) {
                block_cache[j] = 0;
            }
        }

        block_cache[curr_pos % SimpleDisk::BLOCK_SIZE] = _buf[i];
        curr_pos++;
    }
    // Write the last block_cache to disk
    fs->disk->write(block_ids[(curr_pos / SimpleDisk::BLOCK_SIZE)], block_cache);

    // Record the size of the file
    fs->disk->write(inode->block_id, block_ids);
    inode->size = curr_pos;
    Reset();
    return _n;
}
```

bool File::EoF()
This function simply returns whether the current position of the file reaches the end of the file.

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

kernel.C

void exercise_file_system(FileSystem * _file_system)
I changed the STRING1, STRING2 variable to be larger than 512B to test if the file system successfully store files that requires multiple disk blocks.



```
void exercise_file_system(FileSystem * _file_system) {

    // const char * STRING1 = "01234567890123456789";
    // const char * STRING2 = "abcdefghijabcdefghij";
    const char * STRING1 = "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    const char * STRING2 = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Testing

As mentioned above, I changed the STRING1, STRING2 variable to be larger than 512B to test if the file system successfully stores files that requires multiple disk blocks. Following screenshot shows that the file system works successfully for several iterations.