MP7

Completed	@May 4, 2023
Files & media	handout_mp7-1 (1).pdf
Status	Done
Type	MP

Assigned Tasks

Main: Completed
Bonus Option 1: Completed! 🎉
Bonus Option 2: Did not attempt

System Design

There are 4 main elements in this system: the administrative **file system**, the **inodes**, the **blocks,** and **files**

The file system is the administrator, taking care of updating the inodes and blocks accordingly to create and delete new files, along with updatinng and reading from the disk.

Blocks represent sections of memory in the disk.

Inodes are the go between between files and the file system. It holds the metadata of the file including its name, its file length, and which block number the file information is held in.

Our file system can encode and decode the necessary block and inode information to the disk. That is it can both save itself to the disk, and read its saved version.

When creating a file, we find and format a new inode, and associate it with a free block. Deleting a file is the opposite process, we free up both the inode and block.

Reading and writing to and from the disk is simply reading and writing to the disk at the block number associated with that file as stored in the inode.

Code Description

FileSystem

Added or updated following class members

- `Inode inodes[MAX_INODES]` - create an array of inodes of max size
- `int numOfInodes` - number of used inodes

FileSystem()

Default Constructor.

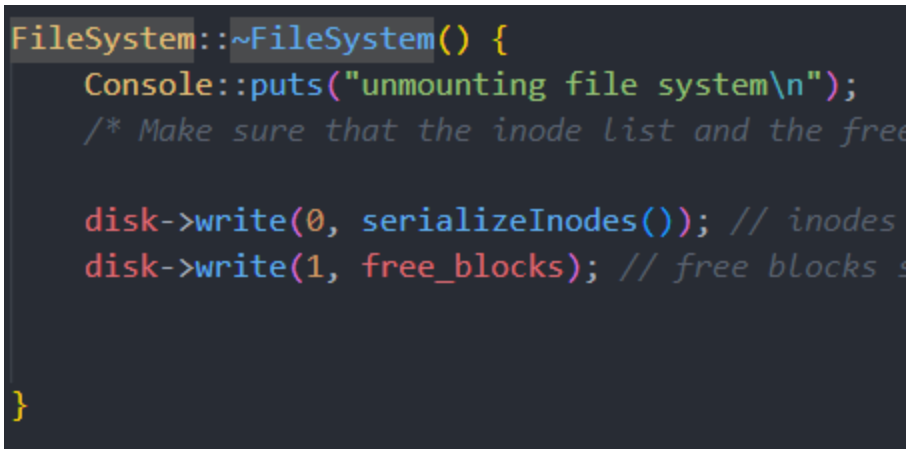
Initialize our class members to default values.

```
FileSystem::FileSystem() {
    Console::puts("In file system constructor.\n");

    disk = nullptr;
    size = 0;
    numOfInodes = 0;
    free_blocks = nullptr;

}
```

~FileSystem()

Destructor.

Write the necessary file system info to the disk; that is, inode and free block information



bool Mount(SimpleDisk* _disk)

Get information from the disk to start our fileSystem

We use deserialize the inodes from the disk block 0, and update `free_block` from disk block 1

```
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;

    unsigned char* buf;
    disk->read(0, buf);

    deserializeInodes(buf);

    unsigned char buf2[SimpleDisk::BLOCK_SIZE];
    Console::puts("");
    disk->read(1, buf2);
    free_blocks = buf2;

    size = disk->size();

    Console::puts("[TEST] Mounting file - inodes: \n");
    for(int i = 0; i < MAX_INODES; i++) {
        assert(inodes[i].free);
    }
    Console::puts("[PASSED]\n");

    return true;

    /*
```

bool Format(SimpleDisk* _disk, unsigned int _size)

Clear the disk + instantiate an empty file system.

Write to our inode block - disk block 0, the long number 0 indicating that it has 0 used inodes. Write to our FREELIST block that every block is free except our inode and free list block.

```
bool FileSystem::Format(SimpleDisk * _disk, unsigned int _size) { /
    Console::puts("formatting disk\n");
    /* Here you populate the disk with an initialized (probably emp
       and a free list. Make sure that blocks used for the inodes a
       are marked as used, otherwise they may get overwritten. */

   unsigned char* buf;
   long end[1] = {0};
   buf = (unsigned char*) end;
   _disk->write(0, buf);


   // see bitmap implementaiton from prev PA
    unsigned char buf2[SimpleDisk::BLOCK_SIZE];
    buf2[0] = '1'; // inode list
    buf2[1] = '1'; // free list
    for(unsigned int i = 2; i < SimpleDisk::BLOCK_SIZE; i++) {
        buf2[i] = '0';
    }

    _disk->write(1, buf2);

    return true;
```

Inode* LookupFile(int file_id)

Find inode with given file_id

Loop through inode list and check if id matches

```
Inode * FileSystem::LookupFile(int _file_id) {
    Console::puts("looking up file with id = "); Console::puti(_fi
    /* Here you go through the inode list to find the file. */
    for(int i = 0; i < MAX_INODES; i++) {
        Inode in = inodes[i];
        if(!in.free && in.id == _file_id) return &inodes[i];
    }

    return nullptr;
}
```

bool CreateFile(int file_id)

Creates a file

Make sure that file with given id does not already exist. Get a free inode and free block. Initialize inode with necessary information. Increment number of Inodes, and mark the free block as taken.

bool DeleteFile(int file_id)

Deletes a file

Ensures file exists in filesystem. Marks both the inode and block as free.

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

    Inode* in = LookupFile(_file_id);
    if (in == nullptr) {
        Console::puts("[ERROR] DeleteFile() - no inode found with id
        return false;

    }

    // free up block no
    int blockNum = in->block_no;
    free_blocks[blockNum] = '0';

    // free up inode
    in->free = true;
    numOfInodes--;

    return true;

}
```

int GetFreeInode()

Returns the index of the next available inode

```
int FileSystem::GetFreeInode() {
    for(int i = 0; i < MAX_INODES; i++) {
        if(inodes[i].free) return i;
    }

    Console::puts("[ERROR] no free inode found\n");
    assert(false);
    return -1;
}
```

int GetFreeBlock()

Returns the index of the next available free block

```
int FileSystem::GetFreeBlock() {
    for(int i = 0; i < SimpleDisk::BLOCK_SIZE; i++) {
        if(free_blocks[i] == '0') return i;
    }

    Console::puts("[ERROR] no free block found\n");
    assert(false);
    return -1;
}
```

unsigned char* serializeInodes()

Encodes necessary inode information to write to disk

For each used node, will add its `id`, `block_no`, and `fileLength` to an array of `long`. At the end, we cast our array of `long*` to `unsigned char*`. Note the very first element in our array is a the number of used inodes.

```
unsigned char* serializeInodes() {
    Console::puts("Beginning to Serialize inodes: \n");
    long* list;
    int counter = 0;

    list[counter++] = numOfInodes;

    for(int i = 0; i < MAX_INODES; i++) {
        Inode* in = &inodes[i];
        if(in->free) continue;

        long temp_id = in->id;
        long temp_no = in->block_no;
        long temp_length = in->fileLength;

        list[counter++] = temp_id;
        list[counter++] = temp_no;
        list[counter++] = temp_length;
    }


    return (unsigned char*) list;
}
```

void deserializeInodes(unsigned char* data):

Decodes disk information that was originally encoded by `serializeInodes` and automatically updates the inodes in the fileSystem.

Cast our `data` to `long*` to correctly process. Loop through the number of inodes encoded, given by the first element of our long* list. For each inode, decode its `id`, `block_no`, and `fileLength` . When adding inodes to our `inodes` list, make sure to also update our `fs` and `free` flag

```
void deserializeInodes(unsigned char* data) {
    Console::puts("Beginning to Deserialize inodes: \n");
    if(data == nullptr) return;

    long* list = (long*) data;


    int counter = 0;

    numOfInodes = list[counter++];

    for(int i = 0; i < numOfInodes; i++) {
        long temp_id = list[counter++];
        long temp_no = list[counter++];
        long temp_length = list[counter++];

        Inode* in = &inodes[i];
        in->id = temp_id;
        in->block_no = temp_no;
        in->fileLength = temp_length;
        in->fs = this;
        in->free = false;

    }

    Console::puts("Deserialized "); Console::puti((int) numOfInode


}
```

Inode

Added following class members

- `unsigned int fileLength` - indicate the total length of the

- `bool free` - flag setting inode as free or taken

Inode()

Default Constructor. Automatically initiate `Inodes inodes[MAX_INODES]`

```
Inode() : id(-1), block_no(99), fileLength(0), free(true), fs(nullptr) {}
```

File

Added class members

- `int currPos` - pointer to where are in the file
- `unsigned long block_no` - which block no does the file reside in
- `unsigned int fileLength` - how long is the file
- `SimpleDisk* disk` - gives file ability to read + write to disk
- `Inode* myInode` - which inode is the file associated with

File()

Default Constructor

Initiate our member variables by finding inode associated with it and reading info from the disk into the `block_cache`

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

    disk = _fs->disk;

    Inode* inod = _fs->LookupFile(_id);

    assert(inod != nullptr);

    block_no = inod->block_no;
    fileLength = inod->fileLength;
    myInode = inod;

    currPos = 0;

    // read file into block cache
    disk->read(block_no, block_cache);


}
```

~File()

Destructor

Writes our `block_cache` onto the disk and updates the appropriate Inode information

```
File::~File() {
    Console::puts("Closing file "); Console::puti(myInode->id); C
    /* Make sure that you write any cached data to disk. */
    /* Also make sure that the inode in the inode list is updated
    myInode->fileLength = fileLength;
    Console::puts("CONTENTS OF FILE: "); Console::puti(fileLength
    for(int i = 0; i < fileLength; i++) {
        Console::putch((char) block_cache[i]);
    }
    Console::puts("\n");
    disk->write(block_no, block_cache);
}
```

int Read(unsigned int _n, char* buf)

Reads _n characters of the file starting at `currPos` into `buf`

Loop starting from `currPos` until `_n` characters have been read or reach the end of the file

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

    Console::puts("current position: "); Console::puti(currPos); Con

    unsigned int counter = 0;
    for(; currPos < _n && currPos < fileLength; currPos++) {
        Console::putch((char) block_cache[currPos]);
        _buf[counter++] = block_cache[currPos];

    }

    Console::puts("\n");

    return counter;

}
```

int Write(unsigned int _n, const char* _buf)

Writes `buf` into our file starting at `currPos` for `_n` characters

Loop starting at `currPos` ending at either the end of the block or after `_n` characters

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

    int counter = 0;
    for(; currPos < _n; currPos++) {
        if (currPos >= SimpleDisk::BLOCK_SIZE) break;
        block_cache[currPos] = _buf[counter++];
    }

    // update the fileLength
    if(currPos > fileLength) fileLength = currPos;


    return counter;
}
```

void Reset()

Resets `currPos` to the beginning of the file

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

bool EoF()

Returns if we are at the end of the file

```
bool File::EoF() {
    // Console::puts("checking for EoF\n");
    return currPos == fileLength;

}
```

Testing

Test 1 - Test Format Function

Test: Test our `Format()` function

Method: At the end of the `Format` function, read in INODE block and FREELIST Block, and print out its contents.

Inode block should just print out 0 (indicates no used inodes)

FreeList block should print out two **1** indicating that the first 2 blocks are taken, and should print out **0** for the remaining blocks.

```
// ****** TESTING format of inode list***********
Console::puts("[TEST] Format of block 0 - inode block \n");
_disk->read(0, buf);
long* l = (long*) buf;
Console::puti((int) l[0]);
assert(l[0] == 0);
Console::puts("[PASSED]\n");

// ****************** TESTING format of free block list *********
Console::puts("[TEST] Format of block 1 - free block \n");
_disk->read(1, buf2);
assert(buf2[0] == '1' && buf2[1] == '1');
for(unsigned int i = 2; i < SimpleDisk::BLOCK_SIZE; i++) {
    Console::putch(buf2[i]);
    Console::puts(", ");
    assert(buf2[i] == '0');
}
Console::puts("\n");
Console::puts("[PASSED]\n");
```

Code

```
formatting disk
[TEST] Format of block 0 - inode block
0[PASSED]
[TEST] Format of block 1 - free block
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
[PASSED]
mounting file system from disk
```

Output

Test 2 - Main Test

Test: Entire system together

Method: Run `kernel.C` with various print statements. Should write and read to 2 different files in an infinite loop


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 2
CONTENTS OF FILE: 20
abcdefghijabcdefghij
Closing file 1
CONTENTS OF FILE: 20
01234567890123456789
Opening file.
looking up file with id = 1
Opening file.
looking up file with id = 2
resetting file
reading from file
current position: 0
01234567890123456789
resetting file
reading from file
current position: 0
abcdefghijabcdefghij
Closing file 2
CONTENTS OF FILE: 20
abcdefghijabcdefghij
Closing file 1
CONTENTS OF FILE: 20
01234567890123456789
deleting file with id:1
looking up file with id = 1
deleting file with id:2
looking up file with id = 2
creating file with id:1
looking up file with id = 1
creating file with id:2
looking up file with id = 2

Bonus Option 1

Overview

To reach our goal of 64KB file with 512B data blocks, we need to allocate (64KB / 512B) = 128 data blocks.

Assuming our disk size is 128KB, we need to be able to map to 128KB / 512B = 2^8 number of blocks. That is, we need an 8 bit number to keep track of all the blocks.

We create an **index block** as a simple array of block numbers. Each entry is 8 bits, or 1 byte long (to track 2^8 blocks). This means, that one index block can point to 512 data blocks.

If our inode points to an **index block** rather than just a singular data block, our inode can keep track of 512 data blocks = 2^18 Bytes = 256KB file size.

Implementation

My FreeList block is manged very poorly at the moment. Per block, I use an entire byte, when only 1 bit is necessary. Thus, I will change my FreeList block management to a bitmap so that one block only uses 1 bit.

Next, my Inode will point to an **Index Block.** That is, allocate a block for an array of block numbers, where each entry is 1 byte. I can make it an `int8_t` array of size 512 so that one entry takes 1 byte.

I will dynamically allocate and deallocate blocks as needed. At any moment, an inode will have *at least* 2 blocks allocated to it - the index block, and the first data block. As I need more and more data blocks, I will begin to add blocks to my **index block** array.

In addition, the cache will be refreshed every time you move into a different data block.

Here is the process for writing a 600 Byte file:

1. Open the file - the inode will indicate where the index block is located, and we are guaranteed that `index[0]` has a ready to use data block.
2. Write cache with information found in the `index[0]` block number from the disk
3. Write to the cache for 512 bytes.
4. Because we hit the block size limit, copy the cache contents into `index[0]` block number into the disk
5. If `index[1]` has not been allocated yet. Get a free block, and add it into `index[1]`. Otherwise, update the cache with the information found in `index[1]` block number from disk.
6. Write the remaining 88 Bytes to `index[1]`
7. On closing the file, write the cache into the `index[1]` data block into the disk.

The reading process is extremely similar to above, but instead of updating any data blocks, we are simply reading them

The only other change is when deleting a file, we will make sure to deallocate blocks as we go down index values.