# CSE 312 Operating Systems Spring 2020
# Midterm Exam Project Report

Ahmed Semih Özmekik
171044039

May 27, 2020

**Abstract**

Demonstration of midterm project homework. Detailed explanation of solution approaches and design choices made for problems encountered in a disk management.

## 1 Problem

A disk will be represented as a 1MB file. When the user opens the disc, he wants to access the data he has saved (just like a user who opens his personal computer). The 1MB disk needs to be managed just as the operating system manages the disk on computers. Of course, the design choices we will make and the operations specified in the homework on the disc are limited. However, as a result of the design to be made, disk operations can be diversified besides these main operations.

In this context, some information should be stored in this file, that is, on disk, in certain offsets on the disk management that will work. We need to decide these offsets, that is, a design on the entire disk in general. Our problems are like this. Now, let's explain the selected designs in detail on the disk.

## 2 Disc Design

As stated in the homework, our design will be quite similar to Unix's design as it uses i-nodes. Let's show the whole disc in the specified parts, and then introduce all the separate chapters in detail. Disc partition sizes in bytes.

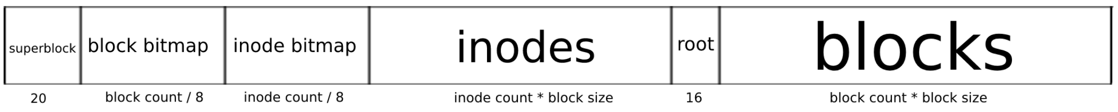

Figure 1: Partitions of Disc

The cursory size and areas in the picture should not be misled. These were made only to provide information on relative sizes. In the picture below, you can see what the size of the partition is in bytes, when block size is 1 KB and inode count is 400.

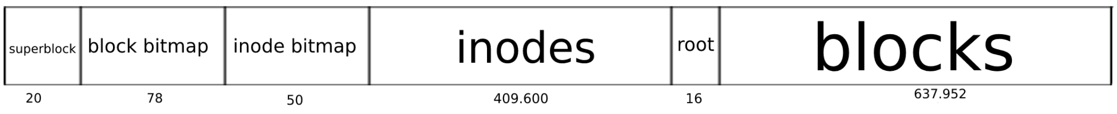

Figure 2: Block size = 1KB, Inode count= 400

As can be seen, partitions vary according to inode size and block size. But regardless, whole disk is 1 MB. So, as the block size grows, the number of blocks will decrease. Or, as the inode count grows, the number of blocks will decrease. The locations of fixed size on the disk are determined. The details of these choices will be specified when we examine each partition individually. Let's move on to introducing the designs of the partitions.

## 2.1 Super-block

As soon as the disc is first opened, the system needs to know some offsets and sizes in advance to restore the disc properly. Super-block contains size offsets that must be known before reading data on disk. As a result, as the offset of the sections shown in Figure 1, offsets will vary according to the block size and inode count, so the system needs to learn them from the beginning and create the data structures accordingly. All this information is stored in super-block. And super-block has a designated area of 10 bytes. Let's examine this fields. (Each partition is 4 bytes long.)

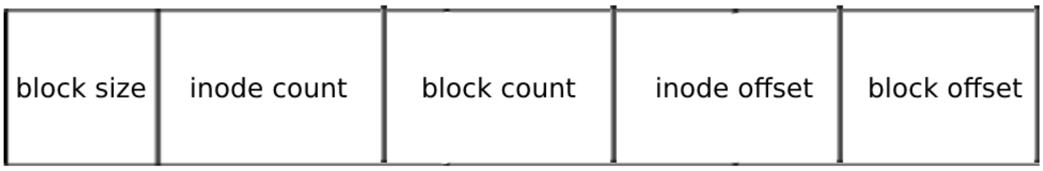

Figure 3: Super-block

## 2.2 Block and Inode Bitmap

It is necessary to state this. There are two different types of blocks in our design: Inode block and data block. The sizes of these two types of blocks are the same. But the difference between these block types is that the number of Inode blocks is certain. The disc is created with a certain number of Inode given by the user. After the arguments are taken from the user, according the amount of space left, the number of data blocks is determined. That's why we have achieved a certain genericity in our design on the similarity of inodes and datablocks. We will further elaborate this generic status when explaining the code-wise details of the design.

The assignment had the following statement: Define how you keep the free blocks and free i-nodes. This problem had to be solved in two different contexts. It is necessary to understand which blocks (both in the inode type and the data block type) are full when reading the disc, and this information had to be observed in a data structure when the disc manager system was running. In other words, we need to solve this problem when the user requests a new block while the system is running and while booting up the system from the disc. We will explain the first one later, now let's explain the disc wise solution.

Our solution is simple and we used the same solution for both data blocks and inode blocks. In these two different block types, a bitmap has been defined according to their total number. In these allocated fields, each bit represents a block index. In this way, while the system is booting, we read this data and pass it to its own data structure so that we can continue where we left off.

## 2.3 Inodes

Since we have come to the Inode section, we can gradually elaborate our design by giving some insight to the application of our designs in the code. As we said, there were two different types of blocks. Let's define the Inode type from these. It was said that "Your file attributes will include size, last modification date and time, and name of the file. No permissions or owner attributes will be kept." We described an Inode structure as mentioned here. You can examine the interface of this structure in the C ++ code below.

```cpp
#define FILENAME_LENGTH_LIMIT 256
class INode : public Serializable
{
public:
    ...

private:
    class FileAttribute : public Serializable
    {
    public:
        ...
    private:
        uint32_t size;
        struct std::tm lastModification;
        char fileName[FILENAME_LENGTH_LIMIT];
    };

    uint16_t *directBlock;
    uint16_t indirectBlock[3]; // 3 block for indirect blocks.

public:
    FileAttribute metadata;
};
```

Functions are hidden to discuss the design more clearly. There are 4 bytes for reserved for size. As in the Unix structure, there is a 256 byte field reserved for keeping the name of the file. 68 bytes are reserved to time. It is 328 bytes in total. According to our account in the whole design, a 2-byte space is sufficient for us to keep the number of the indexes of the addresses in the most efficient way on a 1MB disk.

Therefore, we reserved 2 bytes for inode and data block addresses. There is one single indirect, one double indirect and one triple indirect block in Inode. Since these indirect addresses show a block after all, they make 6 bytes in total to occupy 2 bytes each. Since the block size is not specific to us from the beginning, according to this account, the remaining places from the block size are reserved for direct blocks. In total, 328 + 6 = 334 bytes. Since the rest are, $blocksize - 334$ bytes, there are $\frac{(blocksize-334)}{2}$ direct blocks. In our design, before an inode is created, a block size of an inode is given (in a static variable), and all accounts are shaped accordingly.

Let us also explain the serializable interface. Just like the systems in which MVC, model-view-controller trio is preferred and applied, the system needed a decision structure. A model similar to the MVC structure was considered. After all, we know that Inode structure, Directory and Directory Entry structure, etc. We want to use all the structures we will use the system as follows: Let's save it to Disk and then read it back from the disk and open our object. That's why we want to save our objects to disk in binary format. So, as soon as we read them, we can turn them directly into objects. For this reason, the serializable interface has been defined. But we did not use 100% of this structure of interface and object oriented logic within our system. Even though we wouldn't do that, we followed a nice object-oriented principle here. Logically, we know that whoever wants to be an object that wants to be written to disk and read later should be serializable. Inode, directory and files all defined the write and read functions of serializable in our entire design.

```cpp
/* serializable interface for read and write operations */
class Serializable
{
public:
    virtual char* write(unsigned int*) = 0;
    virtual void read(char*) = 0;
    virtual ~Serializable() {/* */};
};
```

In conclusion, Inodes are defined exactly in the following structure.

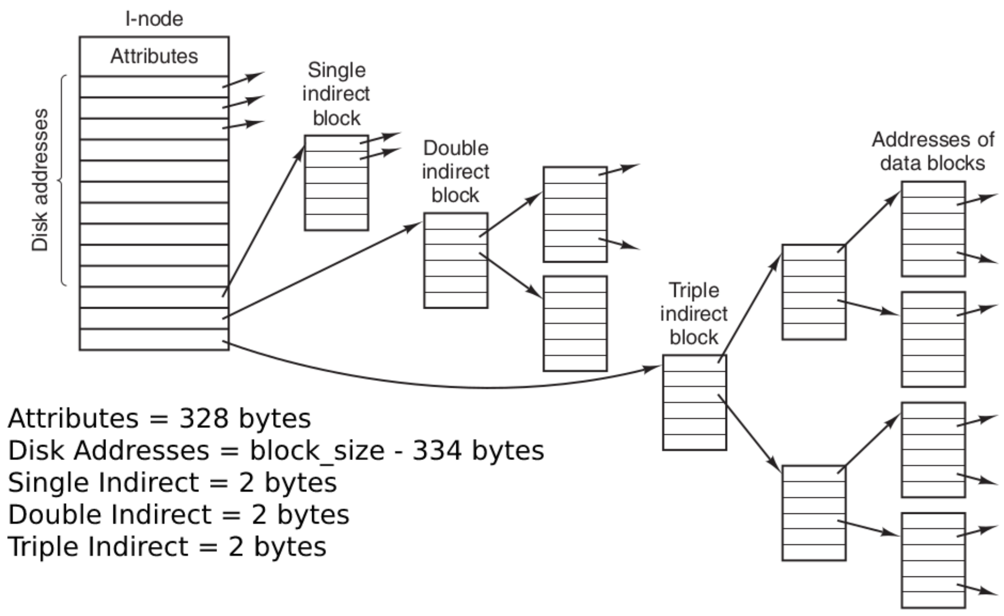

Figure 4: Inode structure

## 2.4 Root

Now that we've got the root partition, we can now define the directory and directory entry configurations.

```
#define FILE_LENGTH 13
class Directory : Serializable
{
public:
    ...

    class DirectoryEntry : public Serializable
    {
    public:
        ...
    private:
        uint16_t inode;
        char fileName[FILE_LENGTH];
        bool isDir;
    };
    ...
private:
    std::map<std::string, DirectoryEntry> files;
};
```

For a directory entry, I based on the following entry, again unix-like.

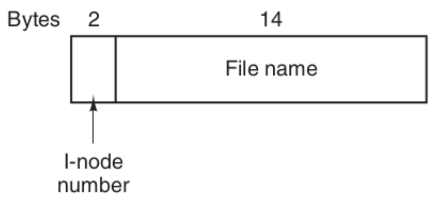

Figure 5: Directory Entry structure

But the only difference is that I took it from 1 byte of the length of the filename and reserved it for a flag that indicates whether the file is a folder or a normal file. You can see this selection in the code. I did this so that the total length is still one of 2, that is, 16.

Directory entry structure is as simple as in Unix. Directory structure is no different from anything that is intended to be recorded in data blocks. Therefore, it does not have a fixed structure. Because a directory is actually a data to be saved in blocks. To get the file we want in constant time complexity, I used a map data structure here. By giving the name of the file, the inode of the file can be accessed instantly.

According to this design, root is saved as a 16-byte directory entry and (when other information is known) offset is known. Thus, the file system always has a root entry and can go to its children whenever it wants. So it actually starts going to all other files from root.

### 2.5 Blocks

There was no need to design a separate object model for blocks. Because a block can be think as a **char** *. Considering that we will break the data into different blocks, there is no better solution than what will be left in a data block being represented by a byte sequence, **char** *.

## 3 Other design approaches

In general, the entire workflow of inodes and blocks revolves around a class called block-manager. There is a helper class called io-communicator used in the block-manager class. Let's try to explain this structure by sharing it below.

```
class BlockManager
{
public:
    /**
     * creates a block manager with given total block size.
     **/
    BlockManager(uint16_t _total, unsigned int size, unsigned int offset
    ~BlockManager();

    void initialize(unsigned int, unsigned int, std::string);
    uint32_t getTotal() const;
    uint32_t freeTotal() const;
    std::set<uint16_t> occupied() const;

    void read(bool *); /* reads bitmap */
    bool *write() const; /* writes bitmap */
    bool inFree(uint16_t);

    /** model: block creator and destroyers */

    uint16_t allocateBlock();
    void deallocateBlock(uint16_t);

    /** controller: block mutator and accessors of the disc. **/

    void setBlock(uint16_t index, void *);
    void *getBlock(uint16_t index);

private:
    std::stack<uint16_t> free;
    std::set<uint16_t> full;
    IOCommunicator communicator;
    uint16_t total; // total block size;
};
```

As can be seen, our design here revolves around two main data structures. The first is a stack structure that holds the addresses that can be assigned in empty, that is, in the next allocation. The second is a set structure that holds the addresses of the blocks in use. There are 2 block manager instances in our system. One for inodes. The other is for blocks.

Block-manager class is a controller class that controls all operations. There is also an io-communicator class that encapsulates disk writing and reading operations. While creating these block-managers, we create them by giving the necessary offset and counts. This is our approach to the aforementioned problem that allows us to know free inodes and blocks (while the system is running).

Our choice here is a not bad design choice for 1MB of disc. Because we can get free inode or blocks in O (1) time complexity thanks to the stack. But of course keeping tarck of occupied ones with **std::set** makes it O(logn) in worst case. **std::set** is commonly implemented as a red-black binary search tree. Insertion on this data structure has a worst-case of O(log(n)) complexity, as the tree is kept balanced. And again, in the set and get operations of the desired block, a find operation is carried out here because we have a look at the set where the occupied are kept for control. This happens with O(log N) time complexity.

So it doesn't seem like a very bad choice. But at any time, as the $set's\ size + stack's\ size = total\ block$, it takes up a lot of memory as the disk grows. But as we said, it's still a nice solution on a 1MB disk.

## 4 Operations

Let's move on to the designs we make in our main class that manages them by using filesystem, all these mentioned classes.

```cpp
class FileSystem
{
public:
    FileSystem(FileSystem const &) = delete;
    FileSystem &operator=(FileSystem const &) = delete;

    static void settings(unsigned int size, unsigned int inodes);

    /**
     * creates a file system to given .data file.
     */
    static FileSystem *create(std::string);

    /**
     * opens a file system from given .data file.
     */
    static FileSystem *open(std::string);

    /**
     * closes the file system.
     */
    static void close();

    /* commands (directly outputed to stdout) */

    void list(std::string);
    void mkdir(std::string);
    void rmdir(std::string);
    void write(std::string, std::string);
    void read(std::string, std::string);
    void del(std::string);
    void dumpe2fs();
    void ln(std::string, std::string);
    void lnsym(std::string, std::string);
    void fsck();
private:
    BlockManager *inodeTable;
    BlockManager *dataBlockTable;
    Directory::DirectoryEntry root;
    static FileSystem *instance;
    ...
};
```

Since the file-system class is a special class, I preferred the singleton design pattern in its design. Creating an instance from the file-system class is only possible in two ways, as seen in the simple interface: This is done by creating a disk from scratch. Or it is done by reading an existing disc. If it is going to be created from scratch, you are given the things to know, which are discussed above with the settings section before creating.

All the operations shared in the homework can be seen on this interface. All that remains is to write an application program that directs the parameters received from the user to the file-system class in the correct order.