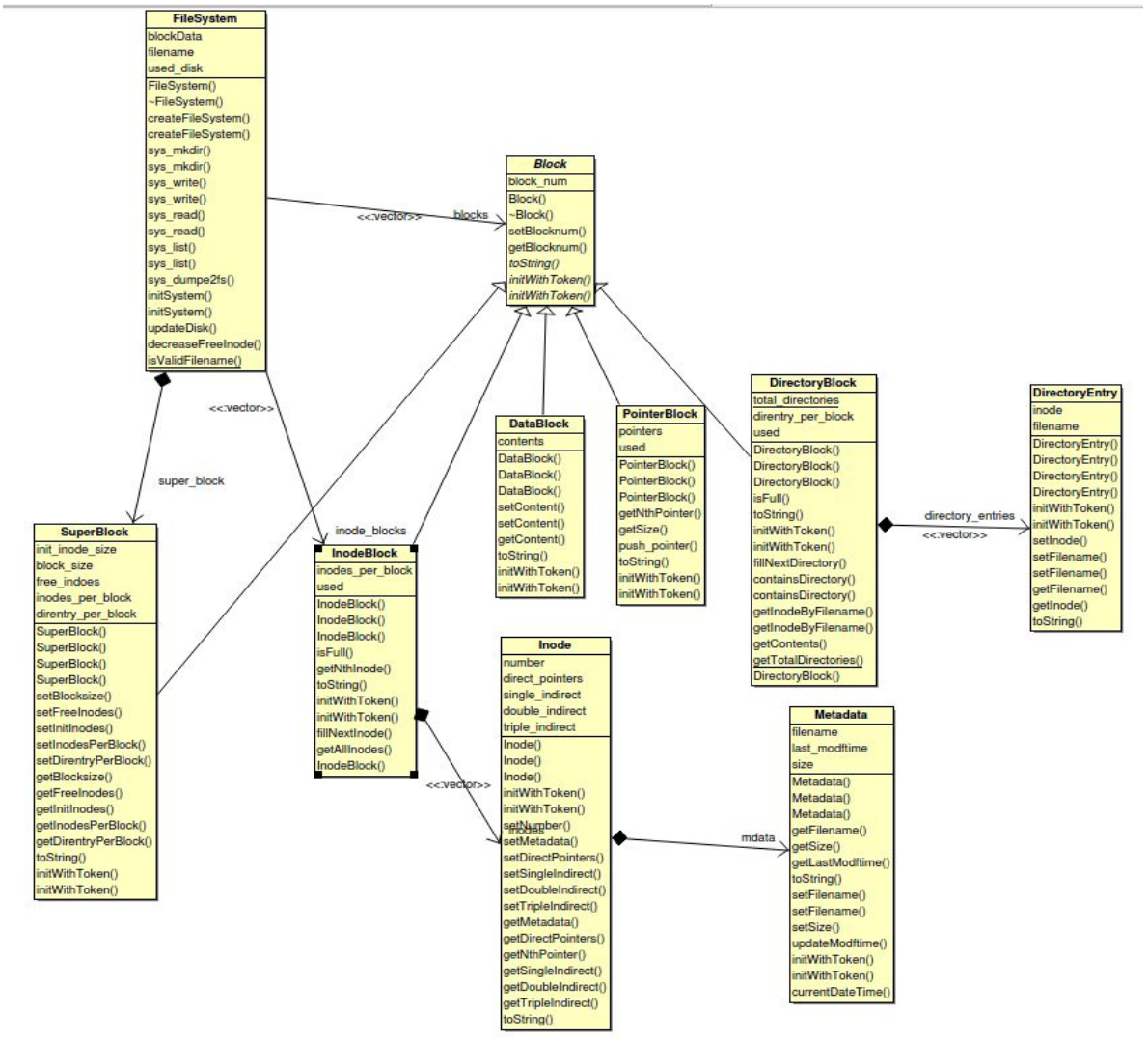
SPRING 2020
CSE 312
OPERATING SYSTEMS

MIDTERM PROJECT
UNIX VARIANT FILE SYSTEM

Design

I used object oriented programming principles to handle data structures that represents the file system components easily.

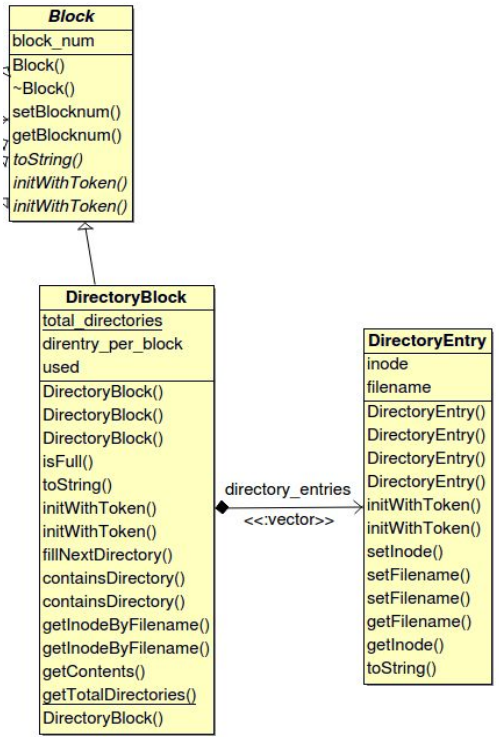
Main perspective of file system design



Directory and directory entry structure

Directory structure represented with **DirectoryBlock** class which derives from the Block parent class. DirectoryBlock contains limited number of directory entries inside it according the block size of the system.

Class DirectoryBlock structure ;



DirectoryBlock class derives from the Block parent class. By this design, it is easy to manipluate the file system.

initWithToken() is abstract at the Block class. Every block structure overrides this virtual method. This function, accepts as a string to construct a block (DirectoryBlock) object. This string is a token got from the disk file that simulates the disk.

Every block, has a special format as a string to write this block objects into the disk. So, Abstract Block class has virtual **toString()** method. This method will be used when the block object written into the disk file.

String format of **DirectoryBlock** class ;

Let's say we have root directory that contains two inner directories which are *usr* and *bin*. Format of the DirectoryBlock will be this ;

[[1,usr];[2,bin]] ---> where [1,usr] and [2,bin] are directory entries.

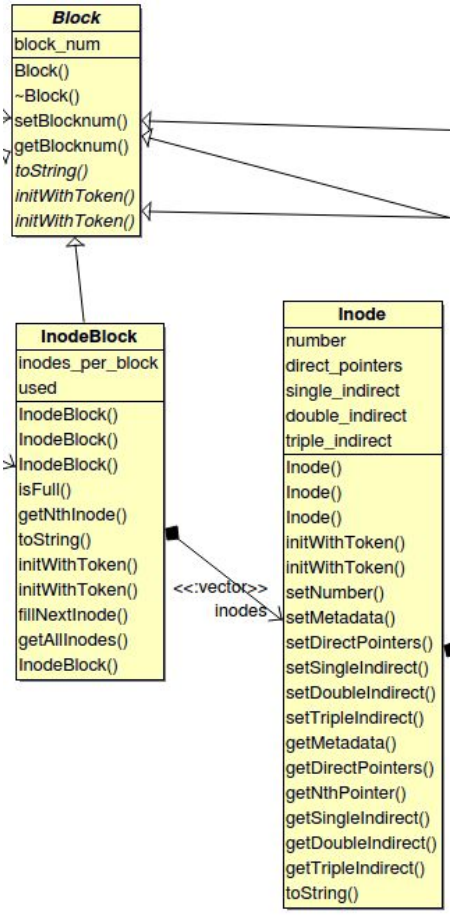
Directory entries are separated with semicolon “**;**”, directory block starts with **[** and ends with **].**

Directory Entry structure

Directory entries are represented with DirectoryEntry class which is described as a diagram in the figure. It has filename and inode number bytes just like in the textbook.

Directory entry has fixed size in my design, which is 32 byte. So, for example, if we have 4KB block size, in one block, there can be maximum 128 directory (or files) inside this block. In my system, the maximum number of directory entries in the directory is calculated by BLOCK_SIZE / ENTRY_SIZE and fixed.

Inode Block and inode structure



InodeBlock class derives from the base Block class. Inode blocks contains array (vector) of inodes. It also contains the used data member. used (int) data member represents the inode block is full or not.

For example, let's say we have 4KB block size. Inode size fixed as 128 byte. This means that, inode block can contain 32 inode inside it.

Inode class represents the inode structure of the file system. It contains 10 direct pointers, one single indirection pointer, one double indirection pointer and one triple indirection pointer.

Note : In my system, double indirection and triple indirection are **NOT** supported. Just single indirection supported.

initWithToken() constructor is used for the same purpose of **DirectoryBlock**. It constructs the object with tokenize which was got from disk file.

This inode structure contains the classical inode data members. It is same with the structure that is in the textbook.

Format of the Inode and Inode block in the disk file. **toString()** method of these classes will give these formats ;

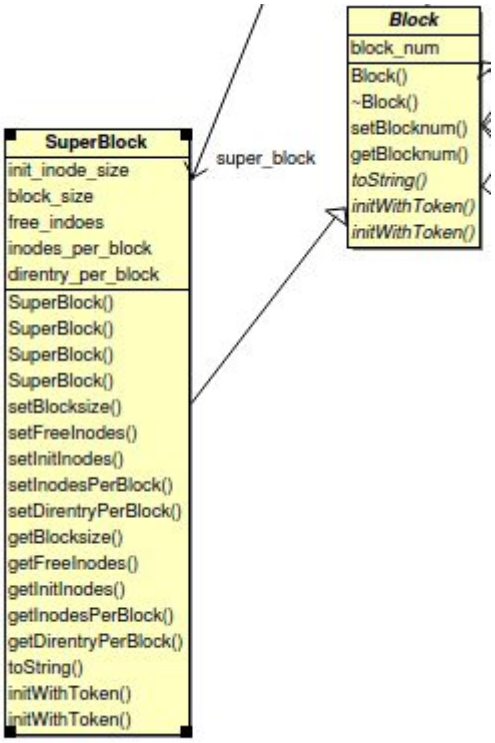
→ Inode :

[<root,4096,2020-05-29.19:36:34>,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
<> part represents the metadata part, other parts represent direct pointers and indirect pointers.
→ first 10 numbers represent direct pointers, remaining 3 pointers are for indirection.

→ InodeBlock :

[[inode1];[inode2];[inode3];] → **";"** separated inodes.

Superblock structure



Super block contains crucial information about the file system. This super block class also derives from the base Block class.

initWithToken() constructor is used for the same purpose of **DirectoryBlock**. It constructs the object with tokenize which was got from disk file.

This superblock contains these informations in order ;

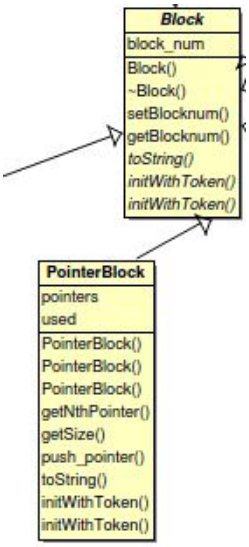
1) Initial inode size
2) Block size
3) Number of free inodes
4) Number of inodes per block
5) Number of directory entry per block

Format of the **SuperBlock** in the disk file. **toString()** method of this will give these formats ;

[4,400,399,32,128]

→ where 4 is the block size as KB, 400 is the initial inode number, 399 is the free inodes, 32 is the number of inodes per block, 128 is the number of directory entry per block. Superblock and dataBlock are separated other blocks, they are special.

Pointer Block structure



Pointer block contains the pointers of blocks. This block is used for indirection. (Single level).

It has simple integer vectors that holds the pointers of data blocks.

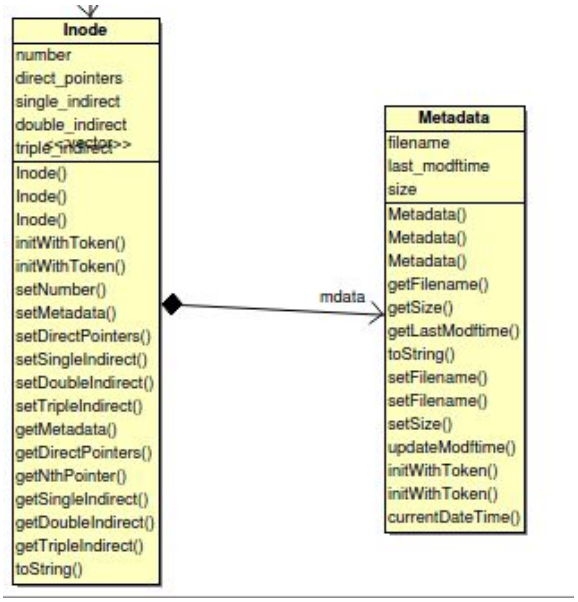
Any inode can points to pointer block for indirection.

initWithToken() constructor is used for the same purpose of **DirectoryBlock**. It constructs the object with tokenize which was got from disk file.

Format of the **PointerBlock** in the disk file. **toString()** method of this will give these formats ;

[1,2,3,4,]
→ where this numbers are pointers the data blocks. (comma separated pointers)

Metadata structure



This metadata class implemented to easy to encapsulate the metadata values in the inodes.

This metadata, is a part of inodes.

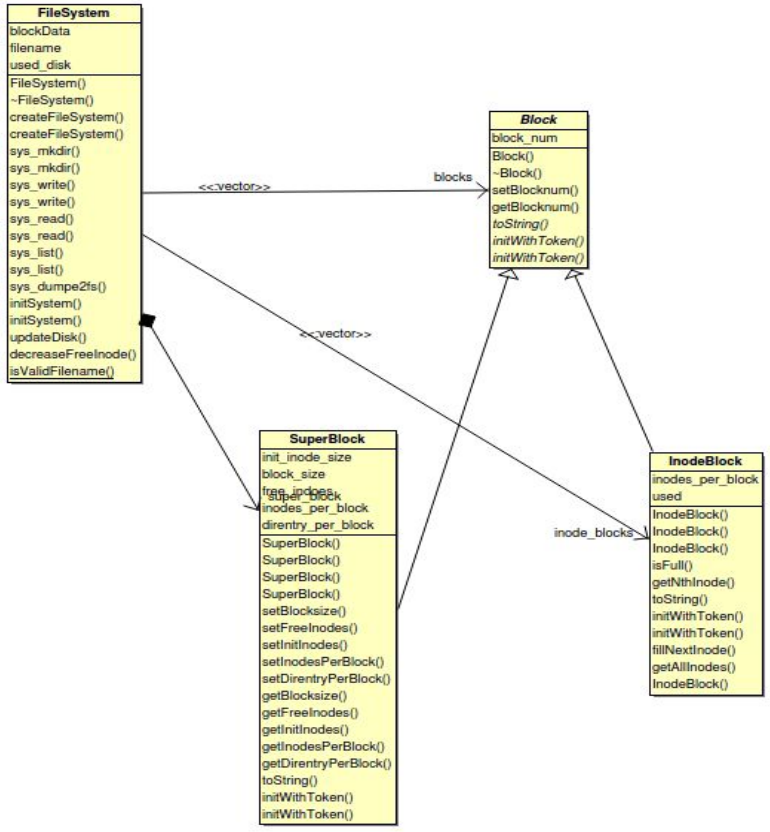
Format of metadata structure, **toString()** method of this Metadata class will give this format ;

<root,4096,2020-05-29.19:36:34>
→ where root is the filename, 4096 is the size as bytes, last one is the modification time.

File System structure

There is a file system class that allows the user interface for the file system. User can initialize the file system by reading disk file and do some operations on it. Main structure of the file system is described in the figure.

Main figure of file system ;



Data members of file system :

- **blockData (string) :** Represents the block types as string. 0 means inode block, 1 means directory block, 2 means data block, 3 means pointer block.
    - For example, if blockData is “012” means that block[0] is inode block, block[1] is directory block and block[2] is data block.

- **filename (string) :** Holds the disk file name. (fileSystem.data)

- **vector<Block*> :** Store the all blocks with Block abstract class pointer through polymorphism.

- **vector<InodeBlock*> :** Store the all inode blocks.

Methods of file operations ; (FileSystem.h and FileSystem.cpp)

- **void sys_mkdir(const char* dirname)**
- **void sys_mkdir(const std::string& dirname)**
- **void sys_write(const char* filename,const char* source_file)**
- **void sys_write(const std::string& filename,const char* source_file)**
- **void sys_read(const char* filename,const char* target_file)**
- **void sys_read(const std::string filename,const char* target_file)**
- **void sys_list(const char* path)**
- **void sys_list(const std::string& path)**
- **void sys_dumpe2fs()**
- **void sys_fsck()**

Private methods ; (not for file operations) ;

- **void updateDisk();**
- **void decreaseFreeInode();**
- **static bool isValidFilename(const char* filename);**

Format of disk file (fileSystem.data)

I will explain the file format on example. Disk file contains 3 lines to store the all necessary information.

1st line stores the super block.
2nd line stores the block data. (defines the type of block)
3rd line stores the all blocks. (can be inode block, data block, directory block, pointer block)

File not contains free blocks and free inodes, just stores the used inodes and blocks. Information of free inodes and free blocks taken at run time.

After we create the file system by **makeFileSystem** program, we get the disk file. (Initially we have root directory and its inode)

[4,400,399,32,128] → Super block data
01 → Block data (says block[0] is inode block, block[1] is directory block)
[[<root,4096,2020-05-29.19:36:34>,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]]~[]~

all blocks are separated with **"~"** character.
block[0] is inode block. It contains one inode for root directory.
block[1] is directory block, It contains the contents of the root directory.

Let's add usr directory under root directory with **mkdir /usr** command. After this operation, disk file will become ;

[4,400,398,32,128]
011 → (block[0] is inode block, block[1] and block[2] are dir block)
[[<root,4096,2020-05-29.19:36:34>,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1];[<usr,4096,2020-05-29.23:17:12>,2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]]~[[1,usr]]~[]~

block[0] is inode block. It contains 2 inodes. One for root directory and one for usr directory. This inodes separated with **";"** character.
→ **First inode** is the root directory inode.
→ **Second inode** is the usr directory inode.

block[1] is directory block. It contains one directory entry which is the usr directory. this directory entry is [1,usr]. It says that, the inode number of the usr directory is 1.

Inode 1 : [<usr,4096,2020-05-29.23:17:12>,2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
Yes, inode 1 contains the usr directory.

2 → this part says that, contents of the usr directory is in the **block 2.**

Additional notes ;

Note 1 : Other implementation details can be found in the source code. I write comments for the methods which are creating empty file system, initializing file system by disk file, file system operations functions etc.

Note 2 : Every class, has specific data format, means every class has specific **toString()** implementation to represent the structures in the disk file. This is like **Serializable** classes in Java. And also, they have **initWithToken()** token method for constructing the classes with tokenize got from the disk file.

Note 3 : My disk contains superblock,block data and all blocks per line, so total 3 lines. Storing the other informations such as block positions is not necessary in my implementation, since all blocks separated with **"~"** character, block positions can be get at the run time, no need for the store them in the file.

Note 4 : In my file system, there is a maximum limit number directory entries in the directories. For example, in 4KB block size system, maximum limit of directory

entries are 128. If directory contains exactly 128 directory entries, mkdir or write operations on this directory is ignored.

Note 5 : In my system, double indirection and triple indirection are not supported. Just single indirection supported. So, if all direct pointers and one single indirect pointer is not enough to store the file, system will use the pointer block that is pointed by single indirect pointer even if it is full or not. It can exceed the maximum block size, but it will continue the using this pointer block. Behaviour is **undefined.** For example, if file is too big to store with 10 direct pointers and one single indirect pointer that points the pointer block. Let's say we have block size of 5 byte. If file is too large, this kind of pointer block can happen : [1,2,3,4,5,6] → this pointer block overflow the block size since it is larger than block size.

Note 6 : After creating file system or doing some operations on the disk, the disk file (fileSystem.data) can be seen in **HEX** format. You can convert it to **UTF-8** again in your editor to increase readability.