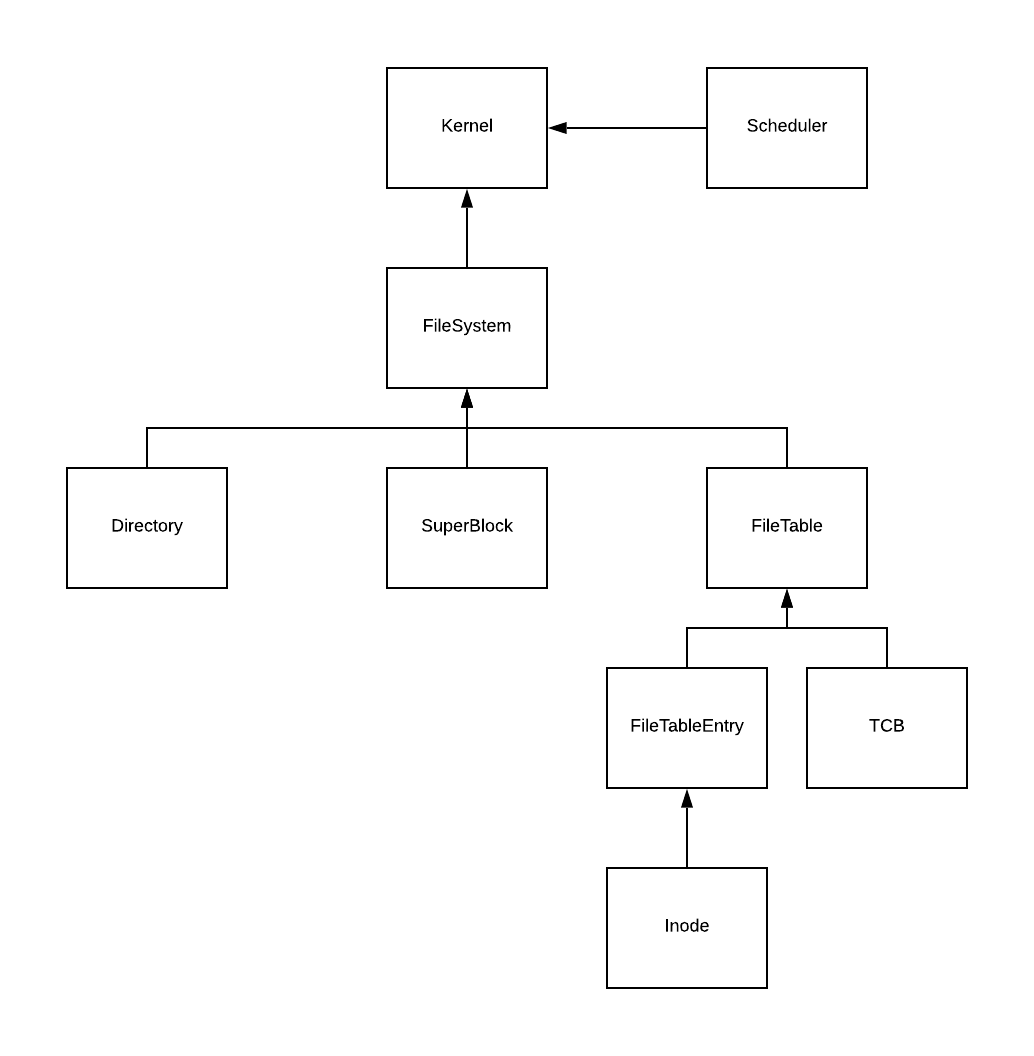
P5 Final Project

CSS 430 A

6/9/2019

Team 17: Cuong Vo, Jessela Budiman, Quan Nghiem

**Design Diagram**



**SuperBlock**

The first disk block, block 0, is called the *superblock*. It is used to describe the number of disk blocks, the number of *inode*s, and the block number of the head block of the free list. The superblock describes the file system and the components. It reads the superblock from disk, and has methods to find free blocks, return free blocks, write, sync, and format.

**SuperBlock()**

superblock constructor which takes in an int for the total number of blocks on disk. Read the superblock and intialize the variables totalblocks, totalinodes, freelist, and inodeblocks.

**sync()**

This method takes the physical SuperBlock data at block 0, and updates the information. Sync will write back to disk the total number of blocks, total inodes, and free list.

**nextFreeBlock()**

This methods finds the first free block available in the free list. The free block is the top from the free queue and returns an integer value for the free block using bytes2int syslib function. If unsuccessful, return -1.

**returnFreeBlock()**

returnBlock method tries to add a newly freed block back to the free list. If the passed in blocknumber is bigger than 0 and less than the total blocks, it will attempt to find the nextfree block. Else, it was unsuccessful and returns false. The newly freed block is added to the end of the free block queue.

**format()**

if the SuperBlock detects an illegal state during initialization of an instance, the method resets the disk of all data and resets correct formatting. SuperBlock variables are reset to the default values and written back to the newly cleared disk.

**Inode**

Each *inode* describes one file. Our *inode* is a simplified version of the Unix *inode*. It includes 12 pointers of the index block. The first 11 of these pointers point to direct blocks. The last pointer points to an indirect block. In addition, each *inode* must include (1) the length of the corresponding file, (2) the number of file (structure) table entries that point to this inode, and (3) the flag to indicate if it is unused (= 0), used(= 1), or in some other status (= 2, 3, 4, ...). 16 *inodes* can be stored in one block.

**Inode()**

this constructor reads the corresponding disk block, locates the corresponding *inode* information in that block, and initializes a new *inode* with this information.

**toDisk()**  
Write back length, count, flag, direct, indirect information of inode back to disk with rawwrite.

**registerTargetBlock()**

run through direct and undirect pointer to register data of pointer is valid and return 0. Otherwise, return error when writing on used block -1, return error when writing on unused block -2, return error on write to null pointer -3.

**setIndexBlock**

return true if the indirect pointer point to to blockID passed, and the data is written.

Return false if Indirect pointer is not -1, All direct pointer is -1.

**findTargetBlock**

Return the target pointer if the target is greater that 11

Return -1 if the target is less than 0

Return the data if byte2short write successfully at blockSpace = (target – directSize) \* 2

**unregisterIndexBlock**

If indirect pointer Is less that 0 return null. Otherwise, read data from indirect pointer. The data is returned to FileSystem to deallocate block.

**Directory**

Directory will store and manage the files within the system. Directory uses two arrays fsize and fname. fsize will hold the sizes of the files. The directory will read data from byte into directory as well as write from directory back to the byte array.  The directory receives the maximum number of *inode*s to be created, and keeps track of which *inode* numbers are in use.

**Directory()**

*Directory* class as the root directory through its constructor, reads the file from the disk that can be found through the *inode* 0 at 32 bytes of the disk block 1, and initializes the  instance with the file contents. Once it receives the directory size, it will create a file name array and sets / as the root directory.

**bytes2directory( )**

This function converts the byte data into integers by a specified offset of a block size. The offset will determine where the block will be placed and is incremented by 4. Each of the file size blocks are set to a name. Traverse and receive each block of characters specified at each location in fsize and assign it to fnames at the index.

**directory2bytes()**

This function initialize the *Directory* instance with a byte array read from the disk and converts the *Directory*instance into a byte array that will be thereafter written back to the disk. The existing file size will be converted for each block into integers which is offset by 4 bytes. From all the file sizes, the data in the files are converted from the directory into the bytes array. A temp string will hold the file names data and the file size, and then convert it to a bytes array. After conversion, the bytes array is copied to the byte array that we are returning.

**ialloc()**

ialloc() finds a file to allocate the filename. It will iterate through the directory to check if there are any empty files with size 0. If found, the new file size and name will be set into the directory. Return location where file was inserted, -1 if unsuccessful.

**ifree()**

This frees up the file size by finding the iNumber in the file sizes array and sets it to 0. If the number is less than the maximum number of characters and the file size at that iNumber is greater than 0 to find it. Else the function returns false and was unable to free it.

**namei()**

This function takes in the file name, and goes through the directory size to find the name to find the node number in that name. If a match is found in the file name length, it will convert the file name location into a string and compared it with the name. Return the index if found, return -1 if unsuccessful.

**printDir()**

Prints out the directory index in the file system and goes through to print out the file size and names at the each index.

**File Table**

Contains the set of file table entries. Each file table entry represents one file descriptor. The class creates file table entries and add it to a Vector of File Table Entries. Remove freed entries from the vector. Naming sense: f stands for files. For example, ffree stands for file free.

**FileTable(Directory directory):**  
Constructor. Create a FileTable vector and set root directory to be the directory from the parameter.

**FileTableEntry falloc (String filename, String mode):**Creates an entry for file. If the file is open in write mode, then only one thread can write in it. Other threads have to wait to read. If another thread wants to write while the file is being written. The thread has to wait until the executing thread finished and closed the file before open and begin writing. Create a file if file does not exit and write mode is called. Increment the number of threads that have used the file.

**boolean ffree (FileTableEntry entry):**Closing and removing FileTableEntry entries from cached list. Check if the last thread that used the FileTableEntry, it wakes up thread with reading status, or all threads if in writing mode.

**Boolean fempty():**Check if FileTable is empty or not.

**FileSystem**  
Performs all the operation on DISK. Implements the basic function of a file system and handle actions like format, open, write, read, delete, seek, and close. Create other classes that compose the project. Act as the interface for all the other classes with the kernel.

**FileSystem (int blocks):**  
Create superblock, directory, and filetable.

**void sync ():**  
Write the data from the directory into the DISK in byte in root directory. Also ensure that the superblock is synced.

**boolean format (int files):**  
Format the disk. Erase all contents of the disk. Then create the superblock, directory, and file tables again. The files parameter is for the number of inodes to be created by the superblock.

**FileTableEntry open (String filename, String mode):**  
Open the file with the corresponding filename. Open the file in the specified mode. Create a new FileTableEntry using falloc(). If it’s in write mode, delete all blocks and write from scratch. Return a FileTableEntry.

**int fsize (FileTableEntry entry):**  
Return the size in bytes.

**int read** **(FileTableEntry entry, byte[] buffer):**  
First check the validity of block to perform read. Read block and calculate buffer size based on data size. Buffer size determines the amount of data read from entry during each iteration.

**int write (FileTableEntry entry, byte[] buffer):**  
Write the contents of buffer to the file corresponded with entry. Called the seekPtr to know where to start writing in file. Increment the pointer by the number of written bytes. Return the number of written bytes or -1.

**private boolean deallocAllBlocks (FileTableEntry entry):**  
check validity of inodes blocks, then run through all registered pointer blocks and invoke superblock.returnFreeBlock. Handles unregistered pointer from inodes. Write inodes back to disk then return true.

**boolean delete (String filename):**  
Delete a specified file. Create a temporary FileTableEntry to contains the inodes. Use the iNumber to free it from directory’s table. Close the FileTableEntry object afterward. Return true if free() and close() are successful.

**boolean close (FileTableEntry entry):**  
Close a FileTableEntry. Return true if successful.

**int seek (FileTableEntry entry, int offset, int location):**  
Updates the seek pointer of FileTableEntry. Returns 0 for success. If pointer is negative or larger than file size, then set to 0 or the file size respectively. Still return successful operation.

**Assumptions/Limitations/ Performance estimation/Current functionality/Possible extended functionality**

The assumptions we made were based on the final project explanations given in the slides, ppt, page, and video. We also made the necessary changes to the Kernel.java files and scheduler. Assuming that all the commands and files are valid and protected. The performance of our program is similar to .class provided and the PowerPoint. It passes all the tests in the Test5.

Limitations to the performance is that we didn't utilize the cache from p4 which would've enhanced the fetching. The original version of each inode is written to the disk every time it changes to keep them consistent. The number of inodes on the disk is limited to 64 based on the assignment page while in file systems, they can handle thousands of files.

Another limitation is our system is complete lack of security for disk and file system because everything is public therefore, it doesn't have any protection.

Another limitation of the performance is our inode only has 11 direct and 1 indirect pointers. Therefore, when we have a file that is large, it can slow down the access process because indirect pointer need to be used to navigate. We can possible use different index implementation, more pointers that can help improve the performance for large file.

The final limitation is that inode is written to disk every time it change. A possible solution for this matter is that we can make it behave like thread that we implemented in P4.

**Successful Results**