Final Project

Final Project: File System

CSS 430: Operating Systems
Date: March 17, 2015
Contributors:

1. Terence Schumacher
2. Nick Abel

Superblock

The superblock is a component of the file system implemented in the CSS430 final project in ThreadOS. It is a block of metadata that describes the file system and its componenets. It reads the physical SuperBlock from disk, validates the health of the disk and provides methods for identifying free blocks, adding blocks to the free list, and writing back to disk the contents of SuperBlock. If validation fails, it will format the disk and write a new SuperBlock to disk

SuperBlock(int numBlocks) — Public constructor for SuperBlock accepts a single int argument equal to the total number of blocks on Disk. The constructor will read the SuperBlock from disk and initialize member variables for the number of blocks, the number of inodes, and the block number of the free list's head.

void sync() — The Sync method brings the physical SuperBlock contents (at block zero on disk) in line with any updates performed to the SuperBlock class instance. Sync will write back to disk the total number of blocks, the total number of inodes, and the free list.

int nextFreeBlock() — The nextFreeBlock method returns the first free block from the free list. The free block is the top block from the free queue and is returned as an integer value. If there is any error (absence of free blocks) -1 is returned to notify the user the operation failed.

boolean returnBlock (int blockNumber) — The returnBlock method attempts to add a newly freed block back to the free list. The newly freed block is added to the end of the free block queue which operates as FIFO. If the freed block does not conform to the actual disk parameters held in SuperBlock, the operation fails and returns false.

void format (int numberOfFiles) — The public format method cleans the disk of all data and resets the correct structure if the SuperBlock detects an illegal state during initialization of an instance. All instance variables of SuperBlock are cleared to default values and written back to the newly cleared disk.

Inode

The main purpose of inode is to describe a file. It holds 12 pointers (11 direct and 1 indirect) of the index block. Inode includes the length of the corresponding file, the number of file table entries that point to the node, and a flag to indicate used or not, plus additional status values. A total of 16 indoors can be stored in a block.

Inode () — Default constructor. Initializes variables and direct/indirect pointers

Inode (short iNumber) — Figures out how many blocks needed. Reads amount of blocks from disk. Initializes data members with buffer size corresponding to the size of each data type. This includes length, count, flag, and the indirect and direct pointers. The blockNumber is calculated by taking the format of 16 indoors per superblock and then adding one to each of the next superblock.

void toDisk(short iNumber) — Write back inode contents to disk. This includes the length, count, flag, direct[], and indirect. Information is saved to the iNumber inode in the disk.

int getIndexBlockNumber(int entry, short offset) — Run through direct and indirect ptrs to block and read data if ptr returns valid, else it will return error code. 0 = unused, -1 = error on write to used block, -2 = error on write on unused block, -3 = error on write to null pointer.

boolean setIndexBlock(short blockNumber) — If index block indirect pointer is not set to -1, or if all the direct pointers are equal to -1 then return false. Else the indirect pointer will point to the blockNumber passed, and data will be written. Returns true if else is the case.

int findTargetBlock (int offset) — If the target is greater than 11 return target block of direct pointer. If the indirect pointer is less than 0 then return the value of the index minus 1. Else, write byte data using bytes2short method at the block space calculated by taking the target block minus the directSize space. Then multiply by 2 for the size of block. Returns data[] of bytes.

bytes[] freeIndirectBlock() — Reads data from indirect pointer unless pointer is value -1, then return null. The data is returned to the FileSystem to deallocate block

Directory

The main purpose of directory is to contain and manage the “files” that are being dealt with. Directory accomplishes this by means of creating two arrays fsize and fname.

fsize is used to contain the sizes of these files in their respective locations. size can be visualized as a simple list of numbers representing the different sizes of file stored int he fname array. Upon Directory’s initialization, the constructor is handed an int called “maxInumber” which is the maximun stored files that the fsize array will hold. Fname is used to contain the “files” that the directory is holding.

The Directory gets broken up into smaller functions to do things like reading data from a byte array into the directory and writing from the directory back to the byte array.

Directory(int maxInumber) — This is the constructor for the Directory class. It receives the desired directory size, initializes all file sizes to 0, creates the file name array and places the "/" rot directory within the first location.

void bytes2directory(byte[] data) — For each entry in the file size, it converts the byte data into integers by a specified offset of a block size. This offset sets the location that each block will be taken from data and is incremented by 4 bytes each time. Each of these blocks is then placed inside of file sizes at the specified index.
From there, we then assign each of the file size blocks with a name, represented by a string temp. Temp holds the entire data block, the total amount of offset, and the maximum bytes allocated. We then obtain each block of characters specified at each location in fsize and assign it to fnames at the specified index.
We don't return anything, but could return 1 for success or 0 for success.

byte[] directory2bytes() — This should be the opposite process from bytes2directory(). First we create a byte array representing the bytes converted from the directory. We take the existing file size and convert each block into integers. This is offset by 4 bytes as well. These coordinate with all file sizes in the directory.
After we've obtained all the file sizes, we can then convert the data in the file names from the directory into the bytes array. We create a temporary string to hold the file names data and the file size, and then convert it to a bytes array. After conversion, we copy the bytes array to the byte array that we are returning. We repeat this process for the size of the directory.

short ialloc(String filename) — This method finds an empty file, and allocates the filename there. It does this by looping through the directorySize and checking to see if there are any 0 file sizes. Once found, it then grabs the new file size, and grabs the characters from the filename and places them into the filenames[]. Once we've completed this we return the location at which the file was inserted, otherwise we throw an error.

boolean ifree(short iNumber) — This finds the iNumber within the file sizes array and sets it to 0. It does a check to see whether or not the number is less than the maximum number of characters and the file size at that iNumber is greater than 0. If either condition returns false the function returns false.

short namei(String filename) — This finds a file iNumber based upon a filename. It checks the file sizes array to see if there's a match in filename length. it then converts the file names location into a temporary string and compares it to the passed in filename. If there is a match, we return the index that contains it. Otherwise we return -1.

void printDir() — This prints out each directory index in the file system and then prints out the file size and file name at that index.

File Table

File (Structure) Table is a class which represents the set of file table entries. Each file table entry represents one file descriptor. The main purpose of this class is to create a new file table entry when it is required and then add that to the Vector of file table entry. It removes the file table entry from Vector when it is freed.

FileTable(Directory directory) — Instantiates FileTableEntry table and sets directory to passed Directory reference

FileTableEntry falloc(String filename, String mode) — Function is responsible for creating a FileTableEntry for requested file. It's ensure, that the file is not opened for write mode for more than one thread and when file is being written other threads cannot read it. When thread wants to open file for write mode, which is already opened of write mode its waiting until previous thread which writes to this file close its FileTableEntry. When file does not exist and is being opened for write mode it is being created. It also increments the number of threads used given the inode as well it is saving created FileTableEntry into internal arrays.

boolean ffree(FileTableEntry entry) — This function is responsible for closing and removing FileTableEntry from cached list. If the thread was the last user of FileTableEntry, it wakes up another thread with reading status, or all threads if writing status occurs.

boolean fempty() — This function returns true if there is not any FileTableEntry cached inside FileTable, and false otherwise.

File System

The file system class is responsible for performing all of the operations on disk. It hides all of the implementation details from users by providing a list of operations which users can directly use. The class implements all the basic functions of a file system as described in lecture, and makes appropriate calls to the components of our system to carry out fundamental actions like format, open, write, read, delete, seek, and close. The file system can be viewed as an API for other files or users to run commands against to access the file system and its contents. The file system has the responsibility of instantiating the other classes that compose our solution.

FileSystem (int blocks) — Creates superblock, directory, and file table. Stores file table in directory.

void sync() — The Sync Method syncs the file system back to the physical disk. The sync method will write the directory information to the disk in byte form in the root directory. The method will also ensure that the superblock is synced.

boolean format(int files) — The format method performs a full format of the disk, erasing all the contents of the disk and regenerating the superblock, directory, and file tables. This operation is not reversible, once formatted all contents are lost. The argument stipulates the number of files (inodes) to be created by the superblock.

FileTableEntry open(String filename, String mode) — This function is responsible for opening a file specified by he filename String passed into it. In addition to the String object, it has passed another String object to represent the mode that the filename object shall have once created. The function starts out by creating a new FileTableEntry object using fileable's falloc()

function. Once that gets created, it checks to see if the mode that was passed in is a “w” for write. If it is, it deletes all blocks and starts writing from scratch. After this check occurs, the new FileTableEntry object is returned to the calling function.

int fsize(FileTableEntry entry) — Returns the file size in bytes atomically.

int read(FileTableEntry entry, byte[] buffer) — Read operation runs atomically. Checks target block to make sure it is valid to read from. Else breaks. Then reads block and calculates the buffer based on data size. The amount of data read during each loop is determined by the buffer size, and it gets read from the entry.

int write(FileTableEntry entry, byte[] buffer) — Writes the contents of buffer to the file indicated by entry, starting at the position indicated by the seek pointer. Increments the seek pointer by the number of bytes to have been written. The return value is the number of bytes that have been written, or -1 upon an error.

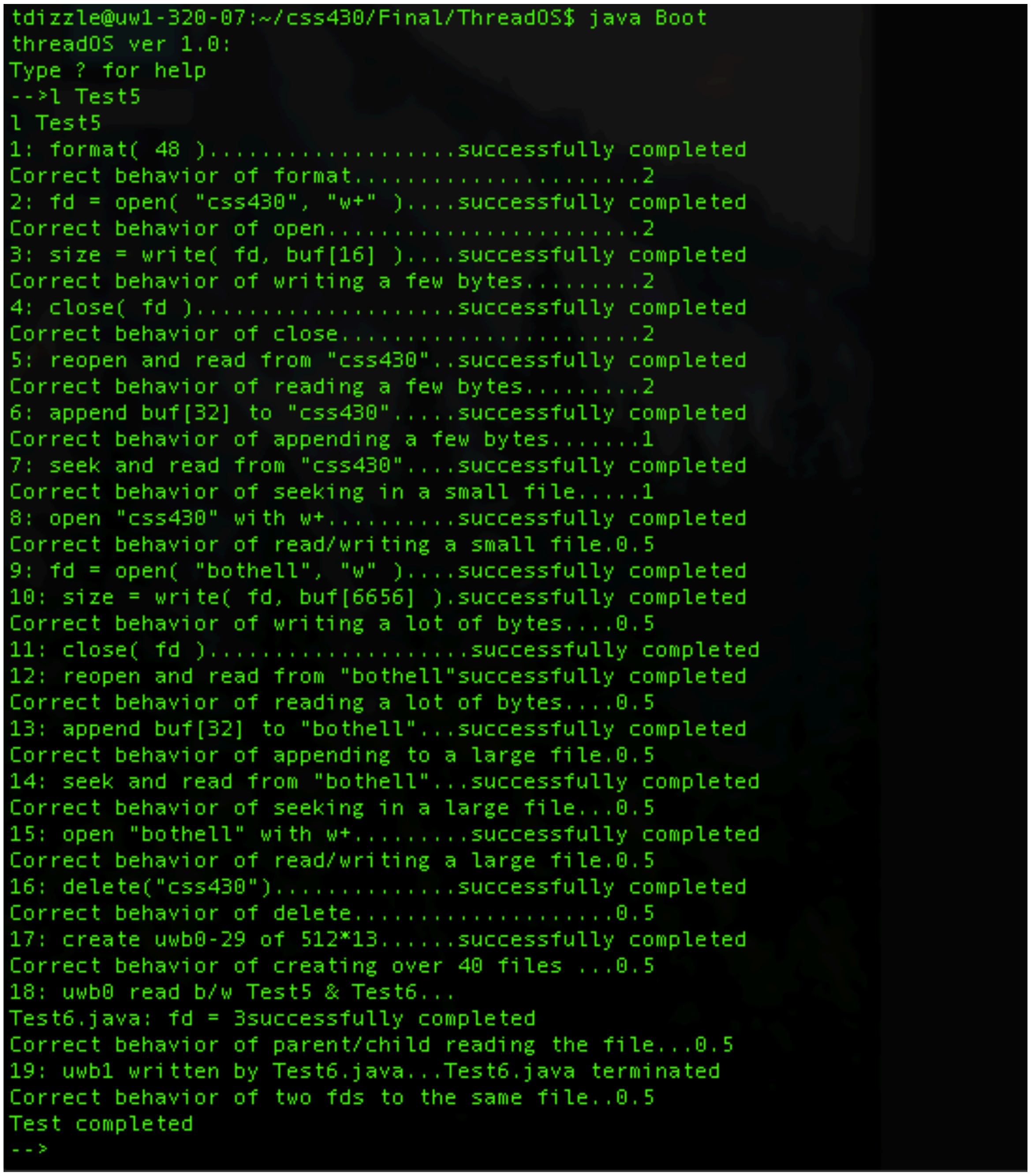
private boolean deallocAllBlocks(FileTableEntry entry) — Checks if inodes blocks are valid, else error. Then runs through all the direct pointer blocks and calls superblock to return if valid. It then handles indirect pointer from inode and calls returnBlock(). It finishes by writing back inodes to disk.

boolean delete(String filename)— This function is responsible for deleting a specified file as per determined by the filename string param passed in. It begins by opening and creating a temporary FileTableEntry object to contain the iNode (TCB) object. This allows us to have access to all private members of this desired filename entry. With this iNode, we use it’s iNumber to free it up from Directory’s tables. Afterwards, we close the FileTableEntry object using the close() function. As long as both the free() and close() are successful, we return true. Otherwise we return false indicating that it is still open elsewhere.

int seek(FileTableEntry entry, int offset, int location) — This function updates the seek pointer corresponding to a given file table entry. It returns 0 if the update was successful, -1 otherwise. In the case that the user attempts to set the seek pointer to a negative number, the method will set it to 0. In the case that the user wants to set the pointer beyond the file size the method sets the seek pointer to the end of the file. In both cases the method returns that the operation was performed successfully.

boolean close (FileTableEntry entry) — This function closes the file corresponding to the given file table entry. It returns true in the case of successful performance of the operation, false otherwise.

Results



```
tdizzle@uw1-320-07:~/css430/Final/ThreadOS$ java Boot
threadOS ver 1.0:
Type ? for help
-->l Test5
l Test5
1: format( 48 )...................successfully completed
Correct behavior of format......................2
2: fd = open( "css430", "w+" )....successfully completed
Correct behavior of open........................2
3: size = write( fd, buf[16] )....successfully completed
Correct behavior of writing a few bytes.........2
4: close( fd )....................successfully completed
Correct behavior of close.......................2
5: reopen and read from "css430"..successfully completed
Correct behavior of reading a few bytes.........2
6: append buf[32] to "css430".....successfully completed
Correct behavior of appending a few bytes.......1
7: seek and read from "css430"....successfully completed
Correct behavior of seeking in a small file.....1
8: open "css430" with w+..........successfully completed
Correct behavior of read/writing a small file.0.5
9: fd = open( "bothell", "w" )....successfully completed
10: size = write( fd, buf[6656] ).successfully completed
Correct behavior of writing a lot of bytes....0.5
11: close( fd )....................successfully completed
12: reopen and read from "bothell"successfully completed
Correct behavior of reading a lot of bytes....0.5
13: append buf[32] to "bothell"...successfully completed
Correct behavior of appending to a large file.0.5
14: seek and read from "bothell"...successfully completed
Correct behavior of seeking in a large file...0.5
15: open "bothell" with w+.........successfully completed
Correct behavior of read/writing a large file.0.5
16: delete("css430")..............successfully completed
Correct behavior of delete....................0.5
17: create uwb0-29 of 512*13......successfully completed
Correct behavior of creating over 40 files ...0.5
18: uwb0 read b/w Test5 & Test6...
Test6.java: fd = 3successfully completed
Correct behavior of parent/child reading the file...0.5
19: uwb1 written by Test6.java...Test6.java terminated
Correct behavior of two fds to the same file..0.5
Test completed
-->
```

Performance, Functionality, Assumptions, and Limitations

Our assumptions in design and implementation are related to the assignment documents provided, namely the powerpoint slides, the pdf document, and the assignment page. We operated under the assumption that the functionality or specifications provided in the assignment are sufficient for the OS and users' needs for file system control. We assumed that all access to files and commands are legitimate, regardless of source, and did not require validation or protection. Additionally, we assumed that file system interaction and instantiation of file system are controlled by test files. The user does not require direct access via the shell and it is sufficient to provide disk commands through compiled Java tests.

As to performance, our implementation performed similarly to the provided .Class files that were developed with ThreadOS. Our observations are qualitative rather than quantitative as the provided test did not include timing functions and using an external source of timing would be insufficient as it could not account for the performance of and interference from the shared lab machines. Like the provided .class files our implementation passed all build validation tests in Test5, and provided identical output as evidenced in the screenshot provided. Additionally, we validated each and every file we created by including them with the precompiled ThreadOS implementation one at a time in isolation. Each file we created to fulfill the requirements of the assignment was validated by completing all portions of the provided test class.

One important limitation on the performance in our implementation of the file system is the fact that the original version of each inode is written to the disk every time it changes, in order to keep them consistent. A possible solution would be keep them in memory and share single instance across all threads. Behaving like a cache, where writes to disk are not necessary unless memory is required, has demonstrated exceptionally better performance in previous labs. Most disk systems, both solid state and diskbased, have some manner of disk caching implemented to improve their performance. We could emulate an an on disk cache with system memory, as the size of the disk in ThreadOS is dwarfed by the amount of available physical memory. In memory, inodes would be kept in an array and saved to disk only when necessary. We might use second chance algorithm, to make sure the hit ratio would be high.

In addition, there are few design issues with current implementation. The first one relates to the limited number of inodes on disk which is 64 (per specification). If we would like to create more, we won't be able. This can be an issue in real file system which should be able to handle thousands of files dynamically. Such an artificially low hard limit on the number of inodes simplified debugging and understanding of disk behavior, but is in stark contrast to modern file system implementations.

Another issue we observed relates to having only 11 direct and 1 indirect pointers in each inode. The direct pointers point to the block disk address where the given block can be found. The indirect pointer points to the indirect block. This solution is very reasonable for files which size do not exceed more than 11 blocks, since it allows direct access to them. However, when the files is large, the indirect pointer has to be used. In that case, the access time is longer since we have to do additional look ups for the disk block which indirect pointer points to. Since we do not have a full understanding of the files our system should be designed for, the solution implemented is adequate. However, other indexing implementations can be better suited for specific applications.

A possible solution (alternative) for that would be keeping blocks' addresses in the linked list. The inode would keep the pointer to the first allocated block for the given file and then each block will keep a pointer to the next block in that file. However, this approach would require to allocate some additional memory in each block for the pointer. Another issue could occur in the case of losing the pointer. In that case the block would be lost because it will be neither in the free block’s list or the content of the file (doublelinked lists can solve this problem, but require additional overhead).

A final limitation on our system is the complete lack of a permission or protection system for the disk and the file system. In a production environment, it is a demonstrated best practice to provide enhanced access control to the file system-protecting files and the system from unauthorized or inappropriate access. Most methods are declared public in our implementation, and any method holding a reference to the file system classes can perform any functioneven those that are normally reserved for elevated privileges.