CSS 430: Operating Systems
Final Project: File System

Professor Robert Dimpsey

March 9th 2020

Parker Amundsen
John Neigel
Sneha Ravichandran

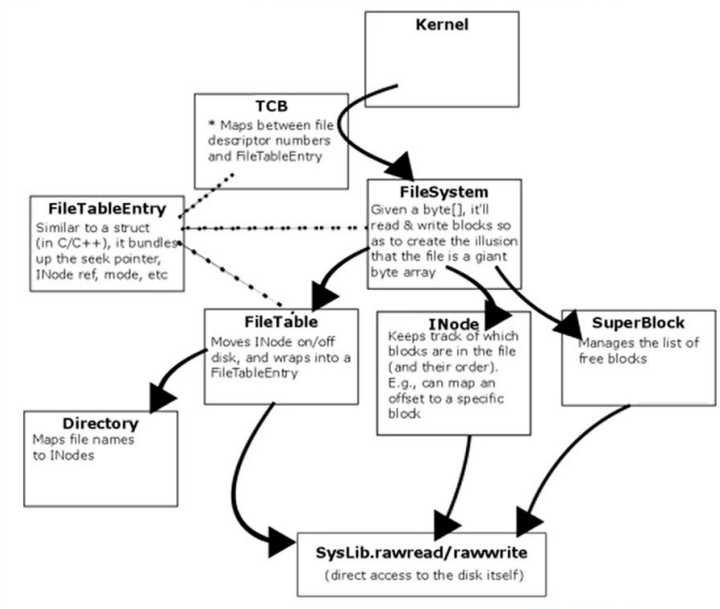
Report Prepared by: Sneha Ravichandran

Table of Contents

Design and Specifications

We followed the design that was outlined in Professor Panitz in his video lecture. Below is a screenshot of his diagram showing what each class does and how they generally interact.

Figure 1: Professor Panitz's Design



To answer the specification questions listed in the assignment, here are the questions and answers listed.

1. **Does the filesystem allow one to read the same data from a file that was previously written?**

Yes, but the work of opening the file is mostly done in the FileTable. The FileSystem just calls the FileTable.falloc(). The FileTable has the code which finds the appropriate file in the blocks and it's respective Inode. Files may be read by multiple readers at a time, but only one writer. And there is no reading while a file is opened by a writer (including w, w+, a).

2. **Are malicious operations handled appropriately and proper error codes returned? Those operations include file accesses with a negative seek pointer, too many files to be opened at a time per a thread and over the system, etc.**

Yes, it is handled in the FileSystem.java. If statements check the passed offset and whence against the file size, to ensure proper seeking. A -1 is returned for improper seek attempts.

3. **Does the file system synchronize all data on memory with disk before a shutdown and reload those data from the disk upon the next boot?**

It should (haven't tested if it actually does, but I'll test here in a bit). When 'q' is passed to the prompt, the loader.class calls sync() on the filesystem which then writes all data from the root directory to the disk.

Kernel.java

We edited the OPEN, CLOSE, SIZE, SEEK, FORMAT, and DELETE cases within the interrupt method of the kernel. The edits we made added the new functions we implemented to create our own file system implementation.

Scheduler.java

We edited the addThread and deleteThread methods to implement reference counting of the files within each thread. Allowing for the file to be taken out of the FileTable when all of the processes are done using it.

FileSystem.java

We created the FileSystem.java file to perform all the operations on the disk. It implements all the functions in the assignment description (format, open, write, read, delete, seek, and close).

public FileSystem(int diskBlocks)

The basic constructor, it takes in a number of disk blocks and initializes the superblock, directory, and filetable with that information.

public void sync()

This function synchronizes the filesystem with the disk.

public boolean format(int files)

This method reformats the file system to support a new number of files. It will lose any files already stored in the current filesystem.

public FileTableEntry open(String fileName, String mode)

This method allows us to open a file. The way it does that is by creating a FileTableEntry object with the file name and mode specified in the parameters. If the mode is “w” then it shows that the file exists and it will deallocate all the blocks associated with this file we’re opening. Other modes will allow for the FileTableEntry object to be added to the file table.

public boolean close(FileTableEntry entry)

This will remove the FileTableEntry object that is given in the parameters from the file table.

public int fsize(FileTableEntry entry)

This returns the size of the file specified in the parameters.

public int read(FileTableEntry entry, byte[] buffer)

This method will allow for the reading from a file. Using the Inode functions, this method will allow for us to read until the end of the file or until the buffer we're reading to is full. The number of bytes read will be returned if this is successful, otherwise it will return -1 if unsuccessful.

public int write(FileTableEntry entry, byte[] buffer)

This function will allow for us to write to a certain file specified by the FileTableEntry object in the parameters. If the mode of the file is read then the function will return an error. Otherwise the data from the buffer will be written into the available blocks and the bytes written will be returned if this operation is successful.

private boolean deallocAllBlocks(FileTableEntry entry)

This uses the Inode method to deallocate all the blocks that are allocated for a certain file.

public boolean freeBlock(int blockNumber)

This uses the superBlock method to free a block.

public boolean delete(String fileName)

This function deletes a file from the FileSystem by opening the file and checking the file's number, if it can be closed successfully, and using the ifree method, if the block where that file sits can be freed.

public int seek(FileTableEntry entry, int offset, int whence)

This function is used to update the seekptr depending on the entry, offset, and whence value accordingly.

FileTable.java

We created the File Table class to provide a representation for a set of file table entries. Each instance of this object represents a single file descriptor. It's made to create a new file table entry when a process accesses a file and add it to the vector of file table entries.

public synchronized FileTableEntry falloc(String fileName, String mode)

This method allocates a new file table entry for the file provided in the method parameters. It allocates space or retrieves and registers the corresponding iNode using the directory. It increments the Inode's count and writes the Inode back to the disk. The return value is the FileTable Entry object if this method is successful.

public synchronized boolean ffree(FileTableEntry entry)

This method is used to locate the FileTableEntry object specified in the parameters. It stores the corresponding Inode state to the disk and removes the entry from the FileTable. If the FileTableEntry object is not in the table this function returns false.

public synchronized boolean fempty()

This function returns true if the current fileTable object is empty and false if not.

Directory.java

We created this class to manage the files that are being used by the processes.

Directory(int maxInodes)

A basic constructor for the Directory class, it initializes the root directory.

public void bytes2Directory(byte data[])

This method converts a byte array from the disk to an array formatted in the Directory class's form.

public byte[] directory2Bytes()

This converts our Directory array back into a byte array that can be stored on the disk.

public short ialloc(String filename)

This method allocates an Inode for a new file to be opened and searches the fsize for the index with the value 0. If there's no Inodes available it will return a -1.

public boolean ifree(short iNumber)

This method will return true if the Inode with iNumber is in the directory. If it exists it will remove it, otherwise it won't do anything and return false, indicating that the Inode is not in the directory.

public short namei(String fileName)

This method returns the iNumber of the Inode associated with the fileName provided to the method. It will first check the size of the files, if it matches it will then check for the name and if no Inode is found the method will return a -1.

FileTableEntry.java

This object represents every file being used by a process in the file system. If there is a file being accessed to perform an action on it, it has a corresponding FileTableEntry object. This object has:

- seekPtr: a file seek pointer
- Inode: a reference to an inode
- iNumber: the inode number
- Count: a count to maintain the number of threads sharing this entry
- Mode: a mode to show whether the file is read, write, read and write, or append

TCB.java

The TCB is the object that holds all of the FileTableEntry information associated with a thread. It takes in a new thread, a tid, and the parent tid to create an instance of the object and has a bunch of getters and setters for the variables associated with the object, which are:

- Thread: the thread associated with this TCB
- Tid: the thread id
- Pid: the parent of this thread's id
- Terminated: whether this thread is terminated or not
- sleepTime: how much time this thread is sleeping for
- ftEnt: the array of FileTableEntry objects associated with the thread

Inode.java

We created the Inode class to keep track of a location within a file so that we can write/read to and from it.

Inode() – Default Constructor

This method initializes all of our global variables. We have:

- Length: contains the file size in bytes
- Count: number of file table entries pointing to this inode
- Flag: a short to tell us if the file is in use (0 if not, 1 if in use)
- Direct: this is an array of direct pointers
- Indirect: this is the index of where the indirect pointers begin

Inode(short iNumber) – Constructor

This method initializes all of our global variables when retrieving the Inode from the disk. We first initialize some variables like a buffer to read into, the block number, and the offset. Then we

read from the file passing in the block number and the buffer to read to. After that we initialize our global variables based on what we read from the file. After that we have a for loop that goes through our direct array of blocks and fills them with the appropriate data.

Void toDisk(short iNumber)

This function is made to save the Inode to the disk as the i-th node. We start out defining a buffer and indirectBuffer to have two places to read things into. Then we set our offset and lock number and change the values of length, count, and flag. After that we do a for loop to pass in all of the direct block data, then we get all the indirect data, copy it, and finally write it all to the block.

boolean setIndexBlock(short iNumber)

This method makes sure that the index of the indirect pointer is valid, if it’s not valid it returns false by going through all of the spaces in the direct array.

public int getTargetBlock(int iNumber)

This method gets the target block with the number iNumber from the direct array. If it doesn’t exist it returns -1.

public int setTargetBlock(int position, short d)

This function is to set the target block using the position and the size of the block. We first get the target block number and search the direct array, and if it’s not found for different reasons we return different error codes for a block already being registered, the block being unused, and the indirect index being invalid. (-1, -2, and -3 respectively).

public byte[] freeIndirectBlocks()

This method frees all the blocks on and after the indirect index and returns the data inside of those blocks.

public boolean deallocateAllBlocks(short iNumber)

This method deallocates the blocks associated with this Inode instance. If there’s only one entry in the filetable with the certain files that are associated with this process. Since if there’s only one reference and this is the thread referring to them, we can delete them from here.

SuperBlock.java

We created the super block to be the place we store a bunch of metadata about the file system components in the first block of the file. It helps us identify where the free blocks are, whether a file exists in the system and if it's not, create a new entry.

SuperBlock(int size)

The basic constructor of the SuperBlock class, creates a new SuperBlock instance.

public void format(int files)

This method was created to format the disk to support the number of files given. Initializing Inodes for all of them, creating a chain of free blocks by writing the next free block number in the first two bytes of a block and lastly calling the sync function to synchronize the current SuperBlock attributes on the disk.

void toDisk()

This method synchronizes the current SuperBlock attributes with the saved SuperBlock attributes on the disk.

public int getFreeBlock()

This method returns the first free block and updates the freeList to get the index of the next free block after reading from the first two bytes of the initial block.

public boolean freeBlock(int blockNumber)

This method clears out the block with the number blockNumber provided in the parameters. It assigns all the bits to 0 and assigns the freeList to the blockNumber and writes the old freeList to the first two bytes of the blockNumber. If this process is successful the method will return true.

Test Results

```
shell[1]% Test5
Test5
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
1: format( 48 )...................Superblock synchronized
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...
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=2)
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
```