Final Design Document

By: Andrew Montgomery, Daniel Yakovlev

Test Results

```
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test5
l Test5
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
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...
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
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
```

Specifications

Many of the specifications for the File System project were determined by the assignment. However, there were some assumptions that were made, and that were factored into the overall design of the file system.

Assumptions

- The methods provided by the assignment description/notes satisfy all the requirements of the file system
- The system calls outlined by the assignment description will be the only ones used for the file system

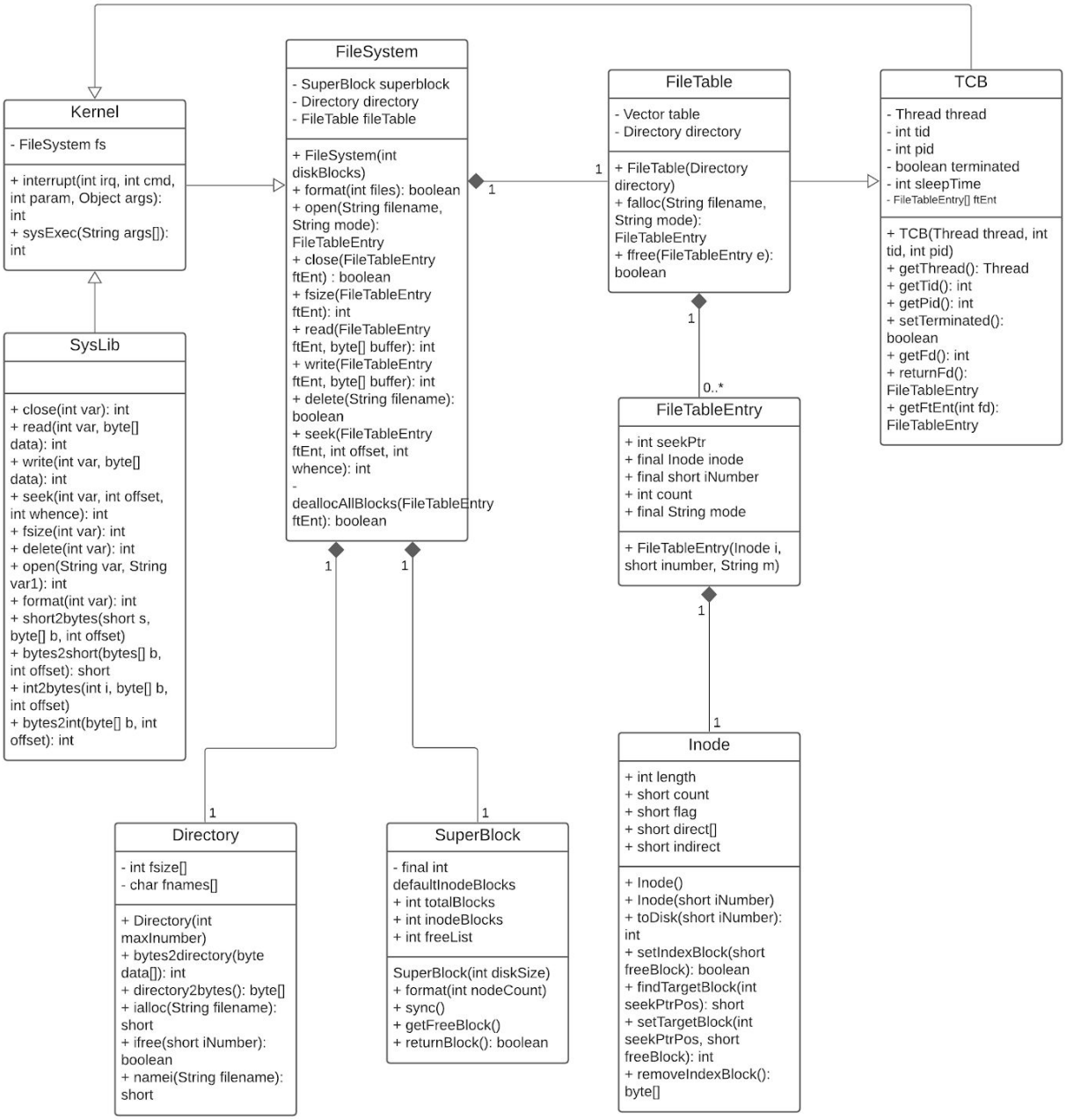
Limitations

- The file system supports only a single-level directory design
- There can only be 29 file table entries per thread
- The Inode size is fixed to 32 bytes
- The Inode supports 11 direct pointers and 1 indirect pointer
- There is no protection in place in terms of data corruption or malicious use
- The File System supports a set number of Inodes (64 spread across 4 blocks)
- File names must be 30 characters long or less
- The maximum number of blocks in each disk is 1000

Design

The File System project was largely a design oriented project. The required classes were provided as part of the assignment specifications, but the overall implementation was largely left to our discretion (though some methods were provided by the notes associated with the assignment). The following sections outline the classes implemented in the File System project and their associated methods.

UML Diagram


Kernel
- FileSystem fs
+ interrupt(int irq, int cmd, int param, Object args): int
+ sysExec(String args[]): int
SysLib
+ close(int var): int
+ read(int var, byte[] data): int
+ write(int var, byte[] data): int
+ seek(int var, int offset, int whence): int
+ fsize(int var): int
+ delete(int var): int
+ open(String var, String var1): int
+ format(int var): int
+ short2bytes(short s, byte[] b, int offset)
+ bytes2short(bytes[] b, int offset): short
+ int2bytes(int i, byte[] b, int offset)
+ bytes2int(byte[] b, int offset): int
FileSystem
- SuperBlock superblock
- Directory directory
- FileTable fileTable
+ FileSystem(int diskBlocks)
+ format(int files): boolean
+ open(String filename, String mode): FileTableEntry
+ close(FileTableEntry ftEnt) : boolean
+ fsize(FileTableEntry ftEnt): int
+ read(FileTableEntry ftEnt, byte[] buffer): int
+ write(FileTableEntry ftEnt, byte[] buffer): int
+ delete(String filename): boolean
+ seek(FileTableEntry ftEnt, int offset, int whence): int
- deallocAllBlocks(FileTableEntry ftEnt): boolean
FileTable
- Vector table
- Directory directory
+ FileTable(Directory directory)
+ falloc(String filename, String mode): FileTableEntry
+ ffree(FileTableEntry e): boolean
TCB
- Thread thread
- int tid
- int pid
- boolean terminated
- int sleepTime
- FileTableEntry[] ftEnt
+ TCB(Thread thread, int tid, int pid)
+ getThread(): Thread
+ getTid(): int
+ getPid(): int
+ setTerminated(): boolean
+ getFd(): int
+ returnFd(): FileTableEntry
+ getFtEnt(int fd): FileTableEntry
FileTableEntry
+ int seekPtr
+ final Inode inode
+ final short iNumber
+ int count
+ final String mode
+ FileTableEntry(Inode i, short inumber, String m)
Directory
- int fsize[]
- char fnames[]
+ Directory(int maxInumber)
+ bytes2directory(byte data[]): int
+ directory2bytes(): byte[]
+ ialloc(String filename): short
+ ifree(short iNumber): boolean
+ namei(String filename): short
SuperBlock
- final int defaultInodeBlocks
+ int totalBlocks
+ int inodeBlocks
+ int freeList
SuperBlock(int diskSize)
+ format(int nodeCount)
+ sync()
+ getFreeBlock()
+ returnBlock(): boolean
Inode
+ int length
+ short count
+ short flag
+ short direct[]
+ short indirect
+ Inode()
+ Inode(short iNumber)
+ toDisk(short iNumber): int
+ setIndexBlock(short freeBlock): boolean
+ findTargetBlock(int seekPtrPos): short
+ setTargetBlock(int seekPtrPos, short freeBlock): int
+ removeIndexBlock(): byte[]
1
0..*

FileSystem.java

FileSystem.java is used to connect all of the different parts of the file system into a cohesive unit that can be used for performing tasks that are necessary for a functioning file system.

FileSystem(int diskBlocks):

Constructor for the FileSystem. Receives the number of disk blocks and creates the file system to be used. The superblock, directory, and the file table are then initialized.

public boolean format(int files):

Method that is used to format the disk. The method erases all data and creates a new directory and file table. The parameter is the number of files that will be formatted. The method returns true when the format is completed

public FileTableEntry open(String filename, String mode):

This method is used to open a file. The method is passed a file name and the mode that the file should be opened in as parameters. The method returns the FileTableEntry where the opened file is located

public boolean close(FileTableEntry ftEnt):

This method is used to close a file. The method ensures that the file is not currently being used elsewhere by checking the count of the FileTableEntry passed as a parameter. The parameter ftEnt is the entry in the FileTable of the file that is being closed. The method returns true if the file was successfully closed, otherwise the method returns false

public int fsize(FileTableEntry ftEnt):

This method is used to get the size of a file. The entry in the FileTable for the file is passed as a parameter. The size of the file in the FileTableEntry is returned if the FileTableEntry exists

public int read(FileTableEntry ftEnt, byte[] buffer):

This method is used to read a file from memory. The method checks to ensure that the file is either in “read” or “read/write” mode. The entry in the FileTable for the file and the data of the file being read is passed as parameters. The method returns an int value representing the total size of the data read from the file

public int write(FileTableEntry ftEnt, byte[] buffer):

This method is used to write data from memory to the disk. The entry in the FileTable for the file and the data being written is passed as parameters. The method returns an int value representing the total size of the data that was written

public boolean delete(String filename):

This method is used to delete a file from the system. The method ensures that the file is closed and that it is removed from the directory. The method is passed the name of the file as a parameter and returns a boolean that determines whether the file was successfully deleted or not

public int seek(FileTableEntry ftEnt, int offset, int whence):

This method is used to set the location of the seek pointer within a given FileTableEntry. The entry in the FileTable of the file, the offset of where the seek pointer should be located, and the location of where the offset starts from are passed as parameters. The method returns the location of the seek pointer after it is moved.

private boolean deallocAllBlocks(FileTableEntry ftEnt):

This method is used to deallocate all blocks associated with the given FileTableEntry. The entry in the FileTable for the file is given as a parameter. The method returns a boolean describing whether the blocks were successfully deallocated or not.

Inode.java

The Inode class is a simplified version of the Linux Inode. Each Inode represents a file system object (i.e. a file) and stores the attributes (length of the file, count of file table entries using the file, flag of whether it is being used or not) of the file. Each Inode has 11 direct pointers pointing to direct blocks, and 1 indirect pointer pointing to an indirect block. Each inode has an iNumber that is used as a unique identifier for the particular Inode. The system avoids Inode inconsistency among different user threads by ensuring that the Inode is read from the disk if it has been updated by another thread before it is updated in memory.

The design for this project also implemented four additional functions that are used for registering, finding, and removing blocks. This is a necessary functionality for the FileSystem.

Inode():

The default constructor for the Inode class. Initializes all the variables to their default values, and sets the direct and indirect pointers to -1, signaling that they are not currently being used

Inode(short iNumber):

Overloaded constructor for the Inode class. This constructor retrieves an Inode from the disk using the iNumber passed as a parameter. The iNumber parameter is a unique identifier used to get the specific block from the disk.

public int toDisk(short iNumber):

This method is used to save an Inode to the disk from memory as the i-th Inode. A unique identifier (iNumber) is passed as a parameter that is used to determine the Inode that is being saved to the disk. The method returns an integer value representing the block number that the Inode is saved to

public boolean setIndexBlock(short freeBlock):

This method is used to set the index block for indirect access. This method only allows the index block to be set when all other direct access blocks are currently in use. The method is passed the free block where the index block will be located. The method returns a boolean that signals whether the index block is successfully initialized or not.

public short findTargetBlock(int seekPtrPos):

This method is used to find the index of a block based upon the position of the seek pointer. The method is used to determine whether a block is available through direct or indirect access. An integer value representing the position of the seek pointer is passed as a parameter. The method returns the location of the target block if it is found, otherwise it returns -1

public int setTargetBlock(int seekPtrPos, short freeBlock):

This method is used to set a target block using the given free block. The position of the seek pointer and the free block that is to be used are passed as parameters. The method returns 0 if the target block is set, otherwise it returns -1, -2, or -3 for errors related to registering info

Kernel.java and SysLib.java

The Kernel class handles all of the system calls that are generated from SysLib.java. The following system calls were designed for this project. Each system call implemented relates to a method in FileSystem.java.

Read

- Reads a file

Write

- Writes to a file

Open

- Opens a file

Close

- Closes a file

Size

- Returns the size of a file

Seek

- Sets the position of the seek pointer

Format

- Formats the disk

Delete

- Deletes a file

Superblock.java

This class represents the superblock of the File System. The superblock is the first disk block and is used to gold the number of disk blocks, the number of inodes, and the block number of the head of the free list in the system. The superblock is managed by the Operating System.

SuperBlock(int diskSize):

Overloaded constructor for the SuperBlock. The number of blocks that can be supported by the system is passed as a parameter.

public void format(int nodeCount):

This method is used to format all of the blocks. The method sets all of the blocks to free and cannot be undone once it is called. The total number of Inodes in the file system is passed as a parameter

public void sync():

This method is used to sync the superblock data to the first block in the disk. The method is used to ensure that the superblock stays updated (i.e. after a format is completed).

public int getFreeBlock():

This method is used to get the next free block. The method returns an integer value representing the location of the free block.

public boolean returnBlock(int blockNumber):

This method is used to return a block to the head of the free list. The block that is returned becomes the head of the free block list. The parameter is the block number of the block being returned. The method returns a boolean representing if the block was returned to the free list or not

FileTableEntry.java

The FileTableEntry class is used as a data structure for entries in the file table. Each entry tracks the seek pointer location, the Inode associated with the entry, the Inode number, the count of the number of threads sharing this entry, and a string representing the mode of the file in the entry.

FileTableEntry(Inode i, short inumber, String m):

This is the constructor for the FileTableEntry. The constructor is passed as parameters the inode associated with the entry, the inumber of the Inode, and the string representing the mode of the entry.

Directory.java

Holds all of the files that have been created and stores them in an array. The index of the file is the inode number that the file is assigned. Files can be removed from the directory. The directory also stores at the beginning of the disk, after the superblock. It will take up a set amount of space depending on how many files can be created. Each file space takes up 64 bytes, 4 for the int file size and 60 for the file name.

public int bytes2directory(byte data[]):

Goes through the data byte array and turns the array into the directory information. The data is in 64 byte chunks of byte data. The first 4 are is the file size int and then the next 60 are for the 60 characters in the filename. The first 4 bytes are turned into an int where it is stored in the fsize array, then the next 60 are turned into a string and stored in the fname array according to the proper index. The directory will always allocate the necessary size to store all of the files that it could store.

public byte[] directory2bytes():

Turns the directory into a byte array. The size of the file is copied into the data byte array with the use of SysLib.int2bytes. Here, the integer is turned into bytes and copied into the output array. Then the file name is turned from a char array into a string, then turned into a byte array using getBytes(). From here it is copied into the output data with the use of arraycopy.

public short ialloc(String filename):

This function allocates a new filename in the directory. It first starts off by going through the array and trying to find an empty spot with fsize of 0. At the same time, it is comparing filenames to make sure that a file has not already been initialized. If a filename that matches the name of the file that is going to be allocated, the function sets a found variable as true. This way, when the function finishes going through the array, it will only initialize a new file if a file matching the filename has not been found. This is done because there could be a file at the very end of the directory that matches the file that is trying to be created.

public boolean ifree(short iNumber):

Free’s a spot in the directory, sets the fsize at index to 0. This way, the file in the directory is deleted without having to change any other data.

public short namei(String filename):

Looks for the inode number of the filename specified. Goes through the array looking for a directory entry with the same filename. If the filename is found, it will return the inode number of that file.

FileTable.java

The file table is where all of the opened file entries are put into and stored. Once that file is no longer needed, it will be deallocated and removed from the file table.

public synchronized FileTableEntry falloc(String filename, String mode):

Allocates a new file table entry into the file table. It first, finds an inode number for the filename using namei. In a while loop It will see if the inode number is more than 0, if it is more than 0, then a new inode will be created, if the flag is a read, it will set the flag to 3, if it is a write it will set the flag to 4. It will then break out of the while loop. If the inode number is less than 0, it means that a file could not be found. Then it will check if the mode is other than read. If it is, it will create a new file and break out of the loop. The loop is used if a file is already written to. After it finishes with the while loop. The function will then increment the inode count, save the inode to disk and create a new file table entry, push it into the file table vector and return the newly created file table entry.

public synchronized boolean ffree(FileTableEntry e):

Free's up a file table entry, it first creates a new inode using the given entries inode number. It will then try to remove the entry. If the entry is not removed it will return false, if it is removed, then it will see if the entry was a read or write entry, if it was a read entry and the count was 1, it will set the inode flag to used. If the entry was a write entry, it will set the flag also to be used. The function then reduces the count for the inode by 1 and saves the inode to the disk and returns true.

public synchronized boolean fempty():

Checks the table vector if it is empty and returns the result.

Considerations

The performance of our File System seemed to be within the expected range based when comparing it to the class files provided by the assignment. In general, the program completes in a reasonable amount of time.

The entirety of the file system performs fast except when formatting and creating a large number of files. This decrease in performance is likely due to the fact that the File System does

not make use of caching, and thus it is necessary for the disk to be constantly accessed when formatting and creating files. This has a minimal impact when creating only one file, but the performance hit can certainly be seen once we start creating a large number of files in succession.

The design of the Inode class also likely results in reduced performance. The design implemented checks the corresponding Inode on the disk, to ensure that it has not been updated, before an inode in memory can be updated by a thread. This results in a significant number of I/O operations to the disk, and likely could be improved by keeping the information in a data structure that is more accessible to all the threads.

The current functionality is somewhat limited based upon the File System being only a single level. The single level design does not allow for files to have the same names or for multiple users to make use of the file system. By redesigning the file system in another design format (such as a general graph structure), it would greatly increase the functionality of the file system as a whole. All of the previous mentioned limitations would be solved, however it would come at the cost of making the system design much more complex as a whole.