Final Project File System

CSS 430 Operating Systems

Contributors: Jeffrey Murray Jr (1876007, jeffmur@uw.edu)
Connor Shabro (1820456, shabrc@uw.edu)
Camila Valdebenito (1561574, cfvb22@uw.edu)

Date: December 6th 2019

Introduction

The purpose of this project was to design and implement a file system on ThreadOS that emulates a unix-like File System. Our file system will enable user programs to access persistent data on disk by way of stream-oriented files. This report will discuss the internal design, testing results as well as assumptions and limitations of our ThreadOS file system.

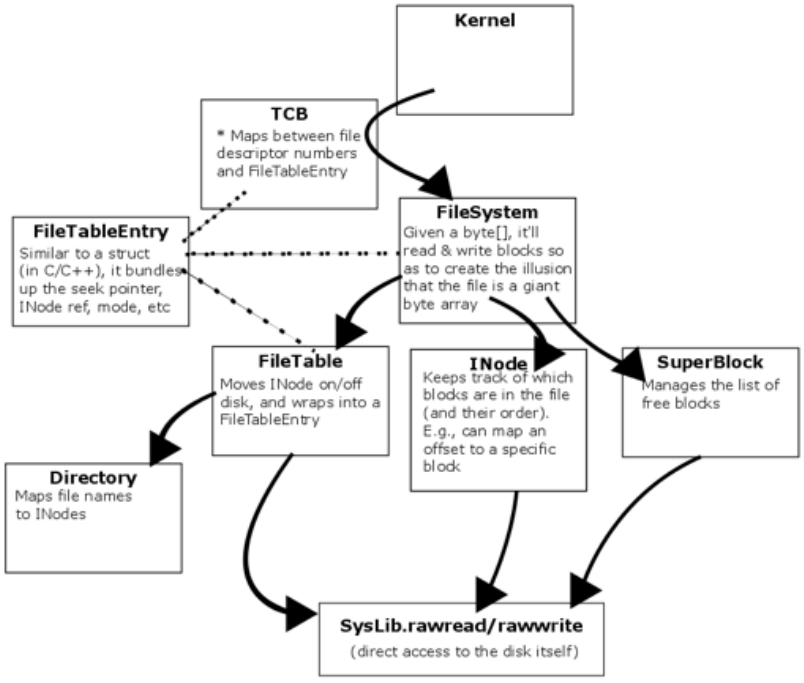

Figure 1. Shows a high-level design of our ThreadOS file system

Implementation

In order to create a ThreadOS file system, our team used the ThreadOS original files provided to us via the Linux Lab. We were also provided *Scheduler.java* and *Kernel.java.* To complete our file system, our team implemented six additional classes: *FileSystem.java, Directory.java, FileTable.java, FileTableEntry.java, Inode.java and SuperBlock.java.*

In this section we talk about the classes we have implemented.

1. FileSystem.java

This class provides a user thread with system calls that enables them to format, open, read, write, update the seek pointer, delete, and get the size of their files. Each File System object contains a SuperBlock, a directory and a file table.

int sync( ):

Syncs file system back to the physical disk, by writing the directory info in byte form in the root directory. Syncs SuperBlock.

int format (int):

Full format of the disk, erasing all the content on the disk, then reinitialise superblock, directory, and file tables. Warning this operation is irreversible!

int open(FileTableEntry, string):

Opens given FileTableEntry with mode. If the file is write, unallocate all blocks, then will wait for the file to be released by another thread.

int close (FileTableEntry):

Removes given file from FileTable. See FileTable.ffree(FTE).

int read(FileTableEntry, byte[]):

Reads up to buffer.length bytes from the file indicated by ftEnt, starting at the position currently pointed to by the seek pointer. If bytes remaining between the current seek pointer and the end of file are less than buffer.length, SysLib.read reads as many bytes as possible,putting them into the beginning of buffer. It increments the seek pointer by the number of bytes that have been read. The return value is the number of bytes that have been read, or a negative value upon an error.

int write(FileTableEntry, byte[]):

Writes the content of buffer to the file indicated by @param FTE, then increments the seek pointer by the number of bytes being written. Returns number of bytes written.

int assignLocation (FileTableEntry):

Helper function for handling iNodes return values. See Inode.getFreeBlockIndex(entry, offset) for more information of these error codes.

int seek(FileTableEntry, int, int):

Updates the seek pointer corresponding to fd as follows:

- If whence is SEEK_SET (= 0), the file's seek pointer is set to offset bytes from the beginning of the file.

- If whence is SEEK_CUR (= 1), the file's seek pointer is set to its current value plus the offset. The offset can be positive or negative.
- If whence is SEEK_END (= 2), the file's seek pointer is set to the size of the file plus the offset. The offset can be positive or negative.

bool deallocEntry(FileTableEntry):
Iterates through direct and indirect pointers of given FileTableEntry, checking if all values are valid, sets them -1, then returns them to the superblock.

boolean delete(String):
Deletes the file specified by given fileName. If the file is currently open, it is not destroyed until the last open on it is closed, but new attempts to open it will fail.

int fsize(FileTableEntry):
Returns iNode.length, user facing, and easy to call :)

2. Directory.java

The main purpose of this class is to contain and manage each file it is given. It is composed of two arrays fsize and fname.

1. fsize: used to contain the sizes of these files in their respective locations.
2. fname: is used to contain the “files” that the directory is holding.

Upon a boot, our ThreadOS file system will instantiate our *Directory* class as the root directory in the constructor. Then, the file system will make a call to the *Directory* class to read the file from the disk that can be found through the *inode* 0 at 32 bytes of the disk block 1, and initializes the *Directory* instance with the file contents. Prior to shutdown, our ThreadOS file system will make a call to our *Directory* class in order to write back the *Directory* information onto the disk.

Directory(int maxInumber):
When a Directory object is instantiated, it takes in the desired directory size. It initializes all the file sizes in the fsize[] to zero to represent that the iNumbers are free.

void bytes2directory(byte data[]):
This method takes in an array that represents Directory information in plain byte.

byte[] directory2bytes( ):
This method converts and returns directory information into a plain byte array. This byte array is written back to disk.

short ialloc(String filename):
This method takes in a filename and allocates a new inode number for this filename.

boolean ifree(short iNumber):
This method takes in a filename and deletes it. It deallocates this iNode number. Returns true if this operation was successful; otherwise, it returns false.

short namei(String filename):
This method returns the iNumber corresponding to the given filename.

3. FileTable.java

Purpose: This class represents a set of file table entries. Each file table entry represents one file descriptor or Inode.
Summary: Keeps track of which Inodes are in memory or disk. If you want to open a file and it's not in memory, filetable asks as a cache to keep track.

FileTableEntry falloc (String filename, String mode)

- Creates a FTE for a requested file
- Ensures the following cases:
    - A file is not open for write by more than one thread
    - While a file is open for right other threads can not read file
    - If a file trying to be accessed is currently open, thread must wait until it is released by other users.
    - If the file does not exist, create, then open in write mode

bool ffree (FileTableEntry)

- Closes and Removes FTE from FileTable
- If the thread was the last user (1), wakes up another thread with reading status, otherwise, wake all threads if the file was being written to.

bool fempty ()

- Returns FileTable.isEmpty()

4. FileTableEntry.java (Provided by Prof. Alicherry)

Purpose: An object that contains pertinent information about a file.

FileTableEntry (Inode, Index Number, Mode)

- int seekPtr: where in the file to read/write/append
- Inode inode: reference to its Inode
- short iNumber: inode's number
- string mode: read, write, write +, append to file

5. Inode.java

Purpose: Describes a file

Summary: Contains 12 pointers (11 direct and 1 indirect), file size, # entries that point to the node, and flag of used, unused, read, or write. Based on the Unix file system implementation, that contains “metadata” aka pointers to other file(s) in the directory. Due to how abstract a file is defined as, we must have plenty of space and pointers to accommodate both file type and sub directories.

Inode()

- Int iNodeSize = 32
- Int directSize = 11
- Int length
- Short count
- Short flag
- Short direct[] = new short[directSize]
- Short indirect

Inode(int index Number)

- Creates Inode (32 bytes of data)
- Length, count, flag, Indirect and direct pointers

int toDisk(in iNumber)

- Copies data from Inode and writes to disk

int fetchTarget (int seekPtr)

- Returns blockIndex based on seekPtr, otherwise -1 on error

bool addBlock (short freeBlock)

- Adds data block to next free slot
- Based on length of iNode

bool setIndexBlock(int blockNumber)

- Traverse through indirect and direct to find null (return false)
- Otherwise sets indirect to blockNumber

int getFreeBlockIndex(int entry, int offset)

- Iterates through direct and indirect and reads when valid
- Returns
    - 0 = unused
    - -1 = error on used
    - -2 = error on unused
    - -3 = error on nullptr

byte[] freeIndirect()

- Fetches current then sets indirect data to -1

6. SuperBlock.java

A SuperBlock is the first disk bloc, block 0, which contains the number of disk blocks, the number of inodes and the block number of the head block of the free list.

- Only one SuperBlock object
- Gives available blocks and an error if none
- Maintains freelist
- Figure out head of free list
- If has spot it should update freelist and write superblock back to disk and update head
    - **Int totalblocks**
    - **Int totalnodes**
    - **Int freelist**
    - **Int blockSize**

SuperBlock(int numBlocks)

Sets blockSize and reads to a byte array called superBlock. It then sets the variables based on the variable distances and the read superBlock from earlier. It checks to see if the superblock is formatted correctly. If so it is finished. If not it calls format(64)

sync()

Creates a temp array of bytes based on the blockSize. It then uses the variables of superblock using the temp array and writes temp to the disk to make sure the disk and superblock are synced.

nextBlock()

First it checks to make sure that freeList is in legal bounds before giving the block. If freeList is alright it creates a tempArray based on blockSize. It then reads to the tempArray bassed on freeList. It then creates a temp int that will hold the value of freeList so that freeList can be updated before returning. freeList is then converted based on tempArray using bytes2int. It then returns tempList. If the freeList was not correct in the first place it just returns -1.

returnBlock(int blockNumber)

It makes sure that the blockNumber given is within the boundaries. If it isn't it returns false. If it is it creates a byte array called newBlock that will be the block you're adding back in. It then updates the freeList to be the newBlock added in so that is the first block accessed. It then writes the newBlock and returns true.

format(int numberOfFiles)

First it updates the totalInodes based on the numberOfFiles. It then creates default Inodes for the totalInodes and adds them toDisk. It then updates the freeList to represent the right block. Then it creates tempArray which it loops through a for loop converting the freeLIst into bytes for the tempArray and then writes to the disk based on the freeList and tempArray. It then updates a new tempArray using superblock variables and writes the tempArray to disk to make sure disk and superblock are synced.

Testing

We ran Test5.java to verify that our FileSystem project was executing correctly. Test5.java executes 19 different methods that test the functionality of every feature and system call. In addition, Test5.java checked the data consistency of our file system.

In general, the following edge cases were tested:

- Does the file system allows one to read the same data from a file that was previously written?
- Are malicious operations handled appropriately and proper error codes returned?
- Does the file system synchronize all data on memory with disk before a shutdown and reload that data from the disk upon the next boot?

Discussion

In this section we discuss the assumptions, performance, limitations and functionality of our file system.

Our team gathered requirements of our ThreadOS file system using the specification from the course website as well as the lecture slides. We assume that the requirements provided are an accurate representation of our user's needs. Some of the assumptions we made include:

- Disk size is 512 bytes
- Disk has a maximum size of 64 iNodes
- Users have access to all of the original ThreadOS files
- Disk is fully implemented and we are able to add on top of it
- No use of cache

Next, we measured the performance of our system by counting the number of passed tests. Our ThreadOS file system successfully passed all 19 tests from *Test5.java* (see Figure 2). We compared the output of *Test5.java* using our file system versus the given one in ThreadOS (via .class files). As a result, our output seems to match the output of the original *.class* files stored in ThreadOS. Therefore, we estimate that our file system behaves correctly.

As to limitations, given that *Test5.java* does not provide a way to measure performance of a system using quantitative data (ie. measuring execution time) we can only measure

performance by comparing output. Another limitation of the original design is that the iNode is written to the disk. One way to improve the performance of this system would be to implement cache memory.

Improving the system could be using the cache if correctly implemented with disk writing and reading functions. If cache is implemented it has a chance of lowering the overall time of running the file system functions. Another way of updating the system is adding more functionality with permissions. Implementing a user and files having multiple levels of permissions based on user, group, and system so that the files have better security.

```
jeffmur@ubuntu-tower:~/UWB/430/AA5$ java Boot
threadOS ver 2.0:
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
-->
```

Figure 2. Shows the output after executing Test5.java.