Ethan Crawford

CSS 430

December 4, 2008

Program 4 - FileSystem

# Test Results

## MyTests.java:

[ethancr@uw1-320-31 ThreadOS]$ java Boot

threadOS ver 1.0:

Type ? for help

threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)

-->l MyTests

l MyTests

threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)

MyTests.run(): entering

**format**

verifySBElements

reformat

verifySBElements

formatWithOpenFiles

invalidFormat

**Correct behavior of format......................1**

**openTests**

Open for read (no file)

Create for write

Open for read

Open for read-write

Open for append

Open max fn

Open invalid mode

Open upper-case mode

Open null fn

Open null mode

Open null strings

Open 0-length fn

Open > max fn

openMaxFiles

openConflictingModes

**Correct behavior of open........................1**

**readTests**

roundTripTests

Null input

Zero bytes

One byte from 1-byte file

One byte from 0-byte file

511 bytes from 512 byte file

512 bytes from 511 byte file

513 bytes from 513 byte file

513 bytes from 512 byte file

512 \* 12 bytes

Inode.maxFileSize - 1 bytes

Inode.maxFileSize bytes

Inode.maxFileSize + 1 bytes

invalidReadTests

**Correct behavior of read........................1**

**writeTests**

roundTripTests

Null input

Zero bytes

One byte from 1-byte file

One byte from 0-byte file

511 bytes from 512 byte file

512 bytes from 511 byte file

513 bytes from 513 byte file

513 bytes from 512 byte file

512 \* 12 bytes

Inode.maxFileSize - 1 bytes

Inode.maxFileSize bytes

Inode.maxFileSize + 1 bytes

invalidWriteTests

**Correct behavior of write.......................1**

**seekTests**

SEEK\_SET positive

SEEK\_CUR positive

SEEK\_END positive

SEEK\_SET zero

SEEK\_CUR zero

SEEK\_END zero

SEEK\_CUR negative

SEEK\_END negative

Seek past eof

Seek before eof

Reserved fd

Invalid fd

SEEK\_SET negative

Invalid seek flag

**Correct behavior of seek........................1**

**closeTests**

**Correct behavior of close.......................1**

**fsizeTests**

**Correct behavior of fsize.......................1**

**deleteTests**

**Correct behavior of delete......................1**

**-------------------------------------------------**

**All Tests Pass..................................8**

MyTests.run(): exiting

-->[ethancr@uw1-320-31 ThreadOS]$ exit

## Instructor’s Test5.java:

[ethancr@uw1-320-31 ThreadOS]$ java Boot

threadOS ver 1.0:

Type ? for help

threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)

-->l Test5 64

l Test5 64

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)

successfully 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

-->[ethancr@uw1-320-31 ThreadOS]$ exit

# Specification

## UML Class Diagram



*Figure 1: FileSystem object model with major supporting objects*

## Description

### Overview

**SysLib** exposes filesystem APIs for stream-style reading and writing via file descriptors by application code. The APIs are implemented as interrupt requests wired into **Kernel**. Kernel stores a **FileSystem** object that implements 8 major functions for file manipulation.

### open(), close(), seek(),file descriptors, and forking

In order to perform file open and close, Kernel relies on the **Scheduler**’s knowledge of the calling thread’s Thread Control Block (**TCB**). The TCB contains a *fileDescriptorTable*, an array of **FileTableEntry** objects. When a user thread creates or opens a file, FileSystem retrieves a new FileTableEntry object from the **FileTable** and stores a pointer to this object in the TCB’s file descriptor table. This object provides the linkage between the file descriptor (an index in the file descriptor table) and underlying Inode. The FileTable stores a pointer to the FileTableEntry in a *tableEntries* vector and increments the object’s refcount. Seeking is implemented using a FileTableEntry member variable.

When a parent TCB creates a child process, the child TCB receives a copy of the parent TCB, duplicates the parent’s file descriptor table and increments the ref counts on each FileTableEntry object. This allows descendent threads to share file descriptors. When the thread is later destroyed, the ref counts are decremented and the FileSystem contents are flushed to physical disk using SysLib.sync(). When a FileTableEntry refcount goes to zero, it is removed from the FileTable.

### format()

FileSystem formatting is an interaction between FileSystem, **Superblock**, **BlockManager**, **Inode**, and **Directory** classes. The FileSystem provides *maxFile* count to the SuperBlock and Directory, reserves the appropriate number of disk blocks using BlockManager, and write maxFile empty Inodes. The remaining empty disk blocks are formatted as a linked list. Basic Disk access occurs via the rawread() and rawwrite() functions in SysLib.

### read(), write(), delete() and fsize(): Inode functionality

The **Inode** object implements stream-based file reading and writing and stores file metadata such as file length and current access mode. Internally, it maintains an index of disk blocks representing the file contents. 12 block pointers are stored directly in the Inode. 11 of these pointers point to blocks used to store file data. The 12th pointer points to a block that stores an array of pointers that point to blocks used to store file data. This layer of indirection allows, with an increased performance cost, the ability to store a large file, relative to overall disk size.

Inode implements methods for read, write, and retrieving file size, as well as infrastructure for loading and saving the file and reading and writing the refcount. The refcount is used by FileTable to determine when to close or delete the Inode.

Deleting a file is an interaction between the Directory (in which the file name is freed and the Inode marked for deletion) and the Inode (in which the actual file contents are freed once the Inode refcount goes to zero). If the Inode was in use at the time of deletion, the FileTable may later clean up the Inode once the refcount goes to zero.

### Directory functionality

The **Directory** object reserves space for maxFiles of *maxChars* in length. Directory information is stored in an Inode. Inodes are associated with file names via an index into the filename string array. Deleted files are removed from the directory listing, but the corresponding Inode number (*iNumber*) is not freed until the Inode refcount goes to zero (when all open file descriptors are closed). This enables \*nix-style data-access-after-deletion.

Directory data is persisted to the filesystem in an efficient manner, with an array of integers, each storing the file name length, written just ahead of the file name data.

### Superblock functionality

The **Superblock** object implements the methods required to update **Disk** block 0, AKA, the Superblock. This block stores the total number of disk blocks, total number of Inodes, and the head pointer to the first available disk block. The Superblock is updated on format and any time a block is allocated or freed.

### BlockManager functionality

The **BlockManager** object exposes methods to allocate and free blocks from the entire Disk. It updates the Superblock when the free block head pointer changes. It also knows how to chain Disk blocks together on format.

## Assumptions

1. Maximum number of files cannot be changed without reformatting the filesystem.
2. Deleted files still open for use take up one directory listing.
3. File names longer than maxChars result in an error.
4. Invalid access mode results in an error.
5. Deleting a file that does not exist results in an error.
6. Deleting an open file is supported.
7. Only one thread may open a file for writing.
8. Many threads may open a file for reading.

## Limitations

1. Only sizeof(short) (2^15) disk blocks are available for addressing due to the current implementation of BlockManager.format().
2. Only one directory path ("/") is supported
3. Max files is bounded by the maximum space available to store Inodes (1000 blocks \* 512 bytes/block / 32 bytes/Inode)
4. Max file size is bounded by the maximum number of block pointers addressable by an Inode (267 \* 512 bytes/block).
5. Only one thread at a time may access the disk (serial access)
6. One entire disk block is consumed by the Superblock.
7. Seeking beyond end-of-file or before beginning-of-file is unsupported.
8. A thread may only access 29 simultaneous file descriptors.

# Internal Design Descriptions

## BlockManager.java

Relatively simple class for adding and removing the head of a linked list of disk blocks and formatting disk blocks as a linked list.

## Directory.java

Provides retrieval of iNumber based on file name and file deletion. The main complexity in this class is in the implementation of the flush() and load() methods, used to read and write the Directory contents to and from disk. The file format used to persist directory information consists of an array of integers followed by an array of characters. Each integer contains the length of the string to load at the character array offset, so the offset must be incremented as each integer is read.

## Disk.java

Other than making a private member public, this implementation was unchanged from the original ThreadOS.

## FileSystem.java

This class contains the public interfaces called by Kernel.interrupt(). As such, the methods are hardened against invalid parameters, to avoid passing invalid values deeper into the system. The class also demonstrates a clear division of labor between Directory, FileTable and Inode. FileTable is used for file open and close, to maintain the TCB/FileTableEntry relationship, and seek. Directory is used for delete, and Inode is used for read and write.

To simplify the underlying system, string mode is mapped to flag bitmask and used, in bitmask form, throughout the rest of the classes. The bit flags represent null, read, write, read/write, unlinked, and appended access states. All of these but append affect reading and writing; append only affects the starting position of the seek pointer.

## FileTable.java

This class creates FileTableEntry objects, maintains a system-wide list of FileTableEntry objects and sets file access policy based on current access mode. When FileTableEntry refcounts go to 0, they are removed from the list. When Inode refcounts go to zero, their file access flags are set to null.

This class provides an instance of itself to each FileTableEntry object. This allows a TCB to call back into the FileTable to clean up stale file descriptors, in the event that a thread exits without calling close() for an open file.

## FileTableEntry.java

This class stores attributes such as Inode pointer, iNumber, file access flags, refcount, and a FileTable pointer. It also provides methods for incrementing, decrementing, and getting the refcount.

## Inode.java

This class contains the heart of the FileSystem – the methods for reading and writing data in a steam-driven fashion. Both read and write check current file access flags, to avoid callers from using a file descriptor open in an invalid mode.

The algorithms for both read and write are the most complex of the project. They involve multiple placeholder variables to keep track of the block, the current block offset, and the number of remaining bytes. A helper function called getBlockPtr() maps between “absolute” block pointer index and direct/indirect block pointers. For write, a function called ensureCapacity() commits all required disk blocks prior to writing, freeing committed blocks in the event that the file is too large for the remaining blocks. Write has the added complexity of being able to grow the file size, so this must be accounted for both when writing and resetting the file length.

Each Inode knows how to write itself at a specific iNumber, based on a simple calculation of starting block and Inode size.

## Kernel.java

This existing class was modified to wire in FileSystem interrupts. On boot, a new FileSystem with a default maxFile count of 64 is created if no existing FileSystem is detected.

## Scheduler.java

This existing class was modified to pass the parent TCB to the child TCB. This allows the TCB class to handle file descriptor duplication on construction. The deleteThread() method was also modified to call SysLib.sync(), allowing for saving of TheadOS’s virtual RAM disk to physical filesystem on thread exit.

## Superblock.java

This simple class reads and writes Superblock information to and from disk block 0.

## SysLib.java

This existing class was modified with 8 functions to expose new FileSystem functionality. Constants for the size of various data types (short, char, and int) were also added, and synchronized keywords were added to rawread, rawwrite, and sync.

## TCB.java

This existing class was modified to support file descriptor table (FileTableEntry array) forking, as well as FileTableEntry adding, removing, and peeking. It is within the setTerminated() method that FileTableEntry.release() calls FileTable.ffree() for any non-null file descriptors, to clean up stale file descriptors on thread exit.

# Performance Considerations

Disk I/O consumes the vast majority of the time spent within FileSystem functions; the most effective optimizations will come from limiting and buffering I/O. Potential optimization points include:

1. Buffering disk reads and writes in block-size multiples
2. Using hashtable or similar to increase filename lookup speed
3. Caching indirect pointer block and write it only when necessary
4. Caching free block list in memory
5. Implementing Disk caching/read-ahead during low CPU consumption times
6. Implementing different block addressing scheme (rather than Inode-style indexing)

# Current Functionality

Current functionality is remedial. Although the read and write algorithms are robust, the maximum file size (136,704 bytes) is tiny, files can only be saved to a single directory and file descriptor tables are fairly small. Nevertheless, the file system is usable within the narrow bounds of ThreadOS and appears to be more or less bug-free.

# Extended Functionality

Extended functionality would likely shore up the deficiencies in the current implementation. The most obvious new feature would be recursive directory support, followed by another layer of Inode indirection to allow realistic file sizes and the use of int for disk block pointer instead of short, to allow for larger disks. Other useful features could be increased file name length, simulation of SCSI-type parallel disk access, the ability to seek beyond end-of-file as a way of resizing the file, and an expanded file descriptor array.

# Partner’s Project Contribution Evaluation (if applicable)

I worked on this project solo and did not have a partner.