CSS 430: File System Project

By Bobby Damore & Connie Kim

# Specifications

## File System Implementation

**Table 1. File System Interface and Required Implementation**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **int SysLib.format( int files );** | 1. Formats the disk (Disk.java’s data contents).    1. The files parameter specifies the maximum number of files to be created (The number of inodes to be allocated) in your file system. 2. The return value is 0 on success, and -1 otherwise. |
| **int fd = SysLib.open( String fileName, String mode );** | 1. Opens the file specified by the fileName string in the given mode (where "r" = ready only, "w" = write only, "w+" = read/write, "a" = append). 2. The call allocates a new file descriptor, fd to this file. 3. The file is created if it does not exist in the mode "w", "w+" or "a". 4. SysLib.open must return a negative number as an error value if the file does not exist in the mode "r".     See Figure 1 (below) for clarification. |
| **int read( int fd, byte buffer[] );** | 1. Reads up to buffer.length bytes from the file indicated by the file descriptor fd, starting at the position currently pointed to by the seek pointer. 2. If bytes remaining between the current seek pointer and the end of file are less than buffer.length:    1. SysLib.read reads as many bytes as possible and puts them into the beginning of buffer.    2. It increments the seek pointer by the number of bytes to have been read.    3. The return value is the number of bytes that have been read, or a negative value upon an error. |
| **int write( int fd, byte buffer[] );** | 1. Writes the contents of buffer to the file indicated by fd, starting at the position indicated by the seek pointer.    1. The operation may overwrite existing data in the file and/or append to the end of the file. 2. SysLib.write increments the seek pointer by the number of bytes to have been written. 3. The return value is the number of bytes that have been written, or a negative value upon an error. |
| **int seek( int fd, int offset, int whence );** | 1. Updates the seek pointer corresponding to fd as follows: 2. If whence is SEEK\_SET (= 0), the file's seek pointer is set to offset bytes from the beginning of the file 3. 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. 4. 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. 5. If the user attempts to set the seek pointer to a negative number you must clamp it to zero and **return success**. 6. If the user attempts to set the pointer to beyond the file size, you must set the seek pointer to the end of the file and **return success**. |
| **int close( int fd );** | 1. Closes the file corresponding to fd, commits all file transactions on this file 2. Unregisters fd from the user file descriptor table of the calling thread's TCB. 3. The return value is 0 in success, otherwise -1. |
| **int delete( String fileName );** | 1. Destroys the file specified by fileName.    1. 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( int fd );** | Returns the size in bytes of the file indicated by fd. |

**Table 2. Open System Call Requirements**

|  |  |  |
| --- | --- | --- |
| **Mode** | **File** | **Requirement** |
| r | Exists | Open file for reading |
| Doesn’t exist | Return error (-1) |
| w, w+, a | Exists | Open file for writing |
| Doesn’t exist | Create file |

## Assumptions and Limitations

* ThreadOS knows how to respond when the File System returns an error.
* Disk size is always 1000 blocks.
* Data type sizes are as follows:
  + Short – 2 bytes
  + Int – 4 bytes
* Any given file can only be used by one Thread at a time, whether it be to read or write.

# Internal Design

## Inode.java

**Purpose** –This class represents a file. It acts as a file control block, containing all of the file attributes and references to data. Our Inode implementation holds 11 direct pointers to file data and 1 indirect pointer to another index block for more data. It also contains the length of the file, flag indicating current usage, and the number of file table entries currently referencing it.

**Inode ( short iNumber )** – This constructor uses the given iNumber to read the corresponding Inode from the Disk. Since the Disk is formatted to store Inodes starting at block 1, we can map the iNumber to the exact block number and inner offset to read the corresponding Inode into memory. The file length, reference count, usage flag, direct pointers and indirect pointer are all read into an Inode instance.

**int toDisk ( short iNumber )** – This method used the given iNumber, which we can map to a specific block and offset, to write the contents of the Inode instance back to the Disk. We do this in the same order that we read the data in from the Disk.

**void allocIndirectBlock ( short blockNum)** – This method uses the given block number to set the indirect pointer, and format the block as an index table for more file data. Initially each index contains -1.

**boolean deallocAllBlocks ( short iNum, Superblock superblock )** – This method deallocates all the indirect and direct blocks of the Inode, making sure the superblock adds them to the free list. It maps the iNumber to the specific block and offset, then writes a new empty Inode at that location on the Disk.

## Directory.java

**Purpose** – This class represents the file system directory. It contains the file names and their corresponding iNumber, which maps to the Inode. This is updated anytime a file is created or deleted.

**int bytes2directory ( byte data[] )** – This method is used to read the Directory from the Disk. It iterates through the given data, which contains the directory data in bytes, and converts the bytes to file names with corresponding sizes. It returns the number of bytes read from data.

**byte[] directory2bytes ( )** – This method is used to write the Directory to the Disk. It converts file names and corresponding sizes to a byte array, and returns it for writing to the Disk.

**short ialloc ( String filename )** – This method is used to add a new file name to the Directory. Given the file name, it allocates a new iNumber by iterating through current file name sizes and finding the first slot with no file (size less than or equal to 0). We then save the new file name length (or set it to maxchars if it’s too big), save the file name, and return its iNumber.

**boolean ifree ( short iNumber )** – This method is used to remove a file name from the directory. Using the iNumber, it sets the corresponding file name size to 0, indicating no file name for that iNumber. If there is already no file name for the given iNumber, this returns false.

**short namei ( String filename )** – This method is used to get the iNumber that corresponds to the given file name. It uses a linear search through the Directory to find the matching file name and size. On success, it will return the corresponding iNumber Otherwise, it returns a -1 to show the given file name does not exist in the Directory.

## FileTable.java

**Purpose** – This class represents the system-maintained File Table shared among all threads. Each entry contains the file’s Inode, corresponding iNumber, number of references to the Inode currently in the File Table, the access mode, and the seek pointer which is positioned based on the access mode (EOF for append, front of file for other modes). Methods are synchronized to prevent race conditions while working with entries.

**FileTable ( Directory directory )** – This constructor instantiates a new vector of file table entries and receives the reference of the directory.

**synchronized FileTableEntry falloc ( String filename, String mode )** – This method is used to allocate a new File Table entry for the given file name in the given access mode. It first looks up the filename in the directory to get the corresponding iNumber. Then it will go through a couple of test cases. If it wasn’t found but the mode was set to read, it returns null. If it wasn’t found but mode was set to write, it allocates a new file in the directory and creates a new Inode. If it was found, read the Inode from the Disk and set the seek pointer to the Inode’s file length if the access mode is append. Then, increment the Inode’s count, and write this back onto the Disk. Finally, we use the Inode, iNumber,, seek pointer, and access mode to create a new file table entry in our file table and return a reference to it.

**synchronized boolean ffree ( FileTableEntry e )** – This method is used to remove an entry from the File Table whenever a Thread is done using the corresponding file. If the given entry doesn’t exist in the file table, return false. Otherwise, it decrements the Inode count and makes sure to set its flag to unused if no one else is using it. Then it writes the Inode back to the Disk.

**synchronized boolean fempty ( )** – This method is used to make sure there are no entries in the File Table before formatting. It will return true if the table is empty, and false otherwise.

## Superblock.java

**Purpose** – This class contains data for the File System. Specifically it describes the number of disk blocks, maximum number of Inodes, and stores the block number at the head of the free block list.

**Superblock( int diskSize )** – This constructor attempts to read the superblock from the first block on the Disk. If the superblock data doesn’t match the given diskSize or expected constraints, the Disk is formatted to the given diskSize and default maximum Inodes. Otherwise, the data read from disk is used to initialize the superblock instance.

**void format( int inodes )** – This method will wipe the disk of all data, write the given number of Inodes to the Disk with default values, write free list pointers to every block after the Inodes, and then synchronizes the superblock with the Disk size, number of Inodes, and free list head.

**void synch ( )** – This method writes the superblock’s contents to the Disk. Should only be called after formatting to ensure consistency with superblock instance and data on the Disk.

**void addToFreeList( int block )** – This method is used when deallocating a Disk block. It adds the given block number to the head of the free list, pointing to the previous free list head block.

**int allocFromFreeList( )** – This method is used to allocate a free Disk block. It returns the head block of the free list and moves the head pointer to the next free block.

## FileSystem.java

**Purpose** – This class represents the entire File System for ThreadOS. It encapsulates all the previously mentioned classes, providing one interface for all file related operations. This layer allows Threads to perform file I/O related tasks without worrying about managing the File System.

**FileSystem( int diskBlocks )** – This constructor uses the given disk size to instantiate a new Super Block instance, then uses the Super Block to instantiate a new Directory instance, and then uses the Directory to instantiate a new File Table instance. The Directory is immediately reconstructed to make sure it contains “/” at it’s root, and any previously saved data for file names.

**void sync( )** – This method ensures that the file system is synchronized with the Disk. It writes the Directory instance to the root file on Disk, and then calls Super Block’s synch() to ensure it also writes to the Disk. File Table Entries only exist in memory, so it doesn’t write them to the Disk.

**boolean format( int files )** – This method wipes all the data from the Disk and reformats it using the Super Block. It then goes through the same File System initialization process to create a new directory and file table.

**FileTableEntry open( String filename, String mode )** – This method uses the File Table to allocate a new entry for the given file name in the given mode, and returns a reference to it. See File Table’s falloc for information on how it’s allocated. Null will be returned if trying to open a file for reading that doesn’t exist.

**synchronized boolean close( FileTableEntry ftEnt )** – This method uses the File Table to deallocate the given File Table entry. See File Table’s ffree for information on how it’s deallocated. This returns false if the given entry is not in the File Table.

**synchronized int fsize( FileTableEntry ftEnt )** –This method is used to obtain the file size for the given File Table entry. It returns the entry’s Inode length.

**synchronized int read( FileTableEntry ftEnt, byte[] buffer )** – This method is used to read part of a file into the given buffer. It first uses the File Table entry’s Inode reference to make sure we don’t read while it’s currently being used (essentially acquiring a file lock). Once it is no longer being used, it sets the Inode flag to read to prevent others from using it. The entry’s seek pointer is used to calculate the index offset within the Inode and block offset to start reading at within the file. It then iterates the file data, reading a block at a time from the Disk, and then onto the buffer. With every iteration, it increments the seek pointer for each byte read, increments the index offset, and resets the block offset to the beginning of the next block for reading more data. Once it reaches the EOF or fills the given buffer, it sets the Inode flag to unused (releasing the file lock), and notifies other Threads. It returns the number of bytes read into the buffer. This method is synchronized to allow a file lock to be acquired and released.

**synchronized int write( FileTableEntry ftEnt, byte[] buffer )** – This method is used to write a given buffer into the given file. It first uses the File Table entry’s Inode reference to make sure we don’t write while it’s currently being used (essentially acquiring a file lock). Once it is no longer being used, it sets the Inode flag to write to prevent others from using it. The entry’s seek pointer is used to calculate the index offset within the Inode and block offset to start writing at within the file. It iterates through the given buffer while writing it to the file data blocks. If the seek pointer reaches the end of the file, it allocates another block of data to the Inode. For allocating direct blocks, only one block is grabbed from the free list and set in the Inode. For allocating the indirect block it passes one free block to the Inode for formatting the new indirect block, and another free block for the first indirect index. For each block it writes to, it has to be read in from Disk, written to from the buffer, and written back to Disk. Finally, it increases the file size if it wrote past the end of the file, sets the Inode flag to unused, and writes the Inode to Disk. Once finished, it notifies Threads that the file is available (releasing file lock), and returns the number of bytes written. This method is synchronized to allow a file lock to be acquired and released.

**boolean delete( String filename )** – This method is used to delete a file. It tries to remove the file from the directory. If it doesn’t exist return false. Otherwise, use the corresponding Inode to deallocate all the blocks containing that file’s data, and add them back to the free list. Finally, reset the Inode to default, and write it to Disk.

**synchronized int seek( FileTableEntry ftEnt, int offset, int whence )** – This method is used to set the seek pointer for the given File Table entry. Depending on the value of whence, the given offset may be directly translated to a seek position, added to the previous seek position, or added to the EOF position. The seek pointer is then clamped to stay in the bounds of the file length, and returned.

# Testing Results

Connies-MBP:ThreadOS conniekim$ 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

**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**

# Performance and Functionality Considerations

* Read and write methods use a busy wait to acquire a file lock. This may result in an unnecessary number of context switches to the waiting Thread for it to continuously check the lock condition.
* Anytime a Thread wants to write to a file, each block it writes to must first be read into memory. This requires at least two accesses to the same block for each block it writes to, while reading only requires the one block access.
* Only one Thread can read one specific file at a time. To extend functionality and performance, a similar implementation to counting semaphores could be used. This would allow many Threads to read the same file at once instead of waiting their turn. Of course no Thread would be able to read while another Thread is writing.
* Formatting requires an access to every single block on the Disk, but does use sequential accessing to speed up the process.
* In order to erase a file’s data and write different data to it (truncation) a Thread would currently have to delete the file and then recreate it. That’s the only way to deallocate all the blocks used by a file in this implementation, so in order to improve performance and extend functionality a truncate method could be implemented.