# CSS 430 Final Project - Thread OS File System

## **Group**: After Work Knight Riders

* Bryan Castillo
* Daniel Martin
* Howard Catlin

## Overview

Our group has implemented a basic Unix like file system within Thread OS. The File System allows users to access data using logical file names using stream oriented operations.

## File System Specification

Programmers access the file system occurs through use of system calls typically through the SysLib class in Thread OS.

|  |  |  |
| --- | --- | --- |
| **System Call** | **SysLib** | **Description** |
| FORMAT | **int format(int files)** | Formats the disk by creating the number of files specified by *files*. |
| SYNC | **int sync()** | On a sync operation, the file system should be synced to disk and the logical disk should be synced to the physical disk. |
| OPEN | **int open(String fileName, String mode)** | Opens a file in the given mode:  “r” - Read  “w” - Write  “a” - Append  “w+” - Read and Write  The file descriptor is returned on success or Kernel.Error is returned on error. |
| READ | **int read(int fd**, **byte[] buffer)** | Reads data from the open file descriptor into the given buffer from the current seek position. The number of bytes read is returned. Kernel.Error is returned on error. |
| WRITE | **int write(int fd, byte[] buffer)** | Writes data from the given buffer to the opened file at the current seek position. The number of bytes written is returned, or Kernel.Error on error. |
| SEEK | **int seek(int fd, int offset, int whence)** | Seek to the given offset in the open file relative to whence. The whence parameter can be:  SET (0) - Relative to the beginning of the file.  CUR (1) - Relative to the current position.  END (2) - Relative to the end.  Values can be positive or negative. If the value is past the end of the file, the seek position will be set to the end of the file. If the seek position is negative, it will be set to the beginning of the file. |
| SIZE | **int fsize(int fd)** | Returns the size of the open file. |
| CLOSE | **int close(int fd)** | Closes the open file. |
| DELETE | **int delete(String fileName)** | Deletes the file with the given name. If the file is currently open it will not be deleted and Kernel.Error will be returned. Any other error will also return Kernel.Error. |
| EXIT | **int exit()** | On thread exit the file system will be synced to help ensure that data is saved. |

### File System on Boot

When Thread OS starts up, the File System will be instantiated. The superblock of the disk will be checked for basic integrity. If the superblock does not pass this check, it is assumed that the File System must be recreated. In this case, it will be automatically formatted with a max of 64 file entries.

### File System Entries

There are only 2 types of entries in the logical file system, which are the root directory and regular files. All entries other than the root directory are considered to be in the root directory. The root directory is named “/”.

### Thread Safety

Single system call operations on the file system are intended to be thread safe. Multiple threads are meant to open, delete, and change files, the root directory, and the overall file system.

### File System Limitations

1. Filenames must not be longer than 30 characters.
2. The maximum size of file is 136,704 bytes.
3. There is only 1 directory, which is the root.
4. A request to delete a file will fail if it is open.
5. There is little metadata available for a file. Only the size is known. There is no concept of ownership or permissions. Creation times and modification times are not known.

## Internal Design

### Disk Layout

The default disk size is 1000 blocks and each block contains 512 bytes. The first block of the disk contains the superblock. The blocks after the super block contain inode entries. The number of blocks needed for inodes is dependent on the maximum number of files the file system is configured with. An inode requires 32 bytes of data. This means each block may be hold up to 16 inodes. The remaining blocks on the disk are used for file or root directory data.

### Component: FileSystem

The FileSystem is the main class that all file related operations go through. It is the main orchestration layer for all other internal components and it is responsible for delegating calls to other components in the correct sequence.

The FileSystem contains a single instance of each of the following types:

* SuperBlock
* Directory - The root directory
* FileTable

The high-level operations typically operate on these objects to operate on the SuperBlock and Directory to manipulate the structure of the file system, while the FileTable is used to gain access to instances of FileTableEntry objects. The FileSystem manipulates and queries information about files using FileTableEntries.

### Component: Scheduler and ThreadControlBlock (TCB)

The ThreadControlBlock is responsible for mapping file descriptors to FileTableEntry instances. File descriptors are integers and are used for the public interface to the file system, while FileTableEntry objects are used as an internal representation of an open file.

The ThreadControlBlock maintains an array of entries for FileTableEntries and it has operations for associating FileTableEntries to file descriptors and for releasing file descriptors.

The Scheduler has 2 main points at which it operates on FileTableEntries in each TCB. When a thread is added, the scheduler will copy file descriptors from the parent to the child. This allows the child thread to use the same open files as the parent. When a thread exits, the Scheduler will close any open file descriptors.

### Component: Kernel

The Kernel implements system calls that operate on the file system. These were described in the first section. The system calls it implements in terms of the file system are: OPEN, READ, WRITE, SEEK, CLOSE, DELETE, SIZE, and FORMAT.

The FORMAT and DELETE operations directly call operations on the FileSystem itself, while the other operations work on file descriptors. These other operations must either create a FileTableEntry or find one and operate on it using the TCB. The OPEN call is responsible for creating a new FileTableEntry and will use the TCB to associate file descriptors with FileTableEntries. The remaining operations will look up FileTableEntries from the TCB and then perform operations on FileTableEntries using the FileSystem.

There are other SYSTEM calls which are important for the lifecycle of the FileSystem itself, but are not direct operations and are thought more of events that the FileSystem should hook. Those are:

**BOOT** - The file system will be instantiated when the Kernel boots.

**EXIT** - The file system will save its contents to disk when a thread exits to help guard against data loss on system failure.

**SYNC** - The file system will save its contents on a disk sync. A disk sync is typically done when Thread OS shuts down. This will ensure that data is saved and available when the system comes back up.

### Component: SuperBlock

The SuperBlock is responsible for managing blocks on disk. It tracks the total number of blocks on disk, the free blocks, and the used blocks. It is also responsible for the overall formatting of the disk itself.

The SuperBlock uses raw block number 0 for storing the meta-data about the disk such as the total blocks, maximum number of inodes, and the pointer to the free blocks. Free blocks are stored and managed using a linked list on disk.

The main operations that the SuperBlock supports are:

1. **format**: The format operation will format the disk.
2. **getFreeBlock**: This operation will attempt to find a free block, remove it from the free list, and return the block for use by the file system.
3. **returnBlock**: When a file no longer needs a block for space, this operation should be called to return the block to the free list.
4. **sync**: This operation should be called to sync any internal cached data back to disk.

### Component: Directory

Directory maintains an entry for each file in the fname and fsize arrays. The fname array is two-dimensional and stores the inode number and filename. The fsize array stores the relevant file size at the same index.

The Directory constructor accepts the maximum number of inodes to be created, which corresponds to the maximum number of files that may be open at one time. It creates the fname and fsize arrays based on this value. All entries in file size entries in fsize are initialized to zero. All entries in fname are left uninitialized.

The main operations that Directory supports are:

1. **bytes2directory**: Accepts a byte array containing Directory data stored on disk. Initialized the Directory instance with data from the input array.
2. **directory2bytes**: Coverts and returns the Directory data in the form of a byte array that will be written back to disk.
3. **ialloc**: Accepts a filename as input and allocates an inode number for new file.
4. **Ifree**: Accepts an inode number and deallocates it. The corresponding file will be deleted.
5. **namei**: Accepts a filename and returns the corresponding filename. Returns an error code if the filename is not in the directory.

### Component: Inode

Inode maintains the list of blocks used by a file and the order in which they are used. It includes 11 pointers to blocks that are directly referenced and 1 indirect pointer. The indirect pointer references a block that stores pointers to additional file blocks. In addition, Inode stores the length of the file in bytes, the number of file-table entries that point to the inode, and a flag that indicates the status of the Inode. The flag values are USED, UNUSED, READ, and WRITE, which represent 0, 1, 2, and 3, respectively.

The Inode constructor initializes the length and count to zero, and the flag to USED (1). It creates direct pointer array and initializes all the entries to -1. Lastly, it initializes the indirect pointer to -1.

The main operation that Inode supports is:

1. **toDisk**: Save the i-th inode to disk. Checks for updates on disk, prior to writing back. Writes *length, count, flag, direct pointer* array, and *indirect pointer*.

### Components: FileTable and FileTableEntry

The FileTable class stores FileTableEntry objects in a vector.

The main operations FileTable supports are:

**falloc**: Accepts a filenamemode and returns a FileTableEntry. Creates entries for write-type modes when an entry does not exits. Returns null for a read mode when an entry does not exist.

**ffree**: Accepts a FileTableEntry and removes it from the FileTable vector. Decreases the Inode’s internal counter and changes flag to USED. Writes the inode back to disk.

**fempty**: Returns true if the table is empty, false otherwise.

FileTableEntry stores data about an open file. Each object corresponds one file descriptor. The class stores the location of the file seek pointer, a reference to an inode, the number of the inode, a count of the number of threads sharing the FileTableEntry, and the mode (r, w, w+, or a). If the mode is “a” the constructor will set the seek pointer to the end of the file.

### Additional Components: FileMode, Seek, and FileSystemException

Three additional components were added. FileMode and Seek contain constants related to file modes and seek relative positioning. FileSystemException is used when a fatal error occurs, but the required API does not provide a way to return an error.

## 

## Performance Analysis

There are multiple areas related to the performance of our file system:

### Free Space Management

For managing free space we used a linked list on disk. This data structure allowed for O(1) block allocation and return. It also required a constant amount of memory regardless of the number of blocks on disk.

One potential performance issue with the linked list free space management, is that over time file data may be scattered over the disk, while some other algorithms might allow files to have a lower average distance between logical sequential blocks.

### Random and Sequential Data Access on Files

It was important for our file system to have reasonable access time for both sequential and random access. In order to get fast random access an indexed allocation scheme was used. This scheme allows for O(1) lookup of a raw block identifier for a given offset in a file. This is done by storing offset mappings either directly in the inode or in another block. This does have 1 disadvantage though, which is that a file has a fairly low maximum size.

Sequential access to file data is reasonably fast and is built on top of random access to the raw disk. One note about our file system is that little effort was put in to optimize low distance between successive logical blocks. This means that more disk seeks are performed than necessary.

### Caching

No effort was put into caching or reading data ahead of time for disk operations in the file system implementation itself. For example, during formatting each inode is written to disk even though 16 inodes share the same raw blocks. This means that 16x more writes are done during formatting than required. Another example of poor performance is in the write operation of FileSystem. Each write operation requires that a read be done of the entire raw block before writing due to the fact that the write may not be as large as a block or is not aligned with a block boundary. If 1 byte were written at a time sequentially, this would result in 1024 block IO operations. For a read operation, if a user were to read 1 byte at a time from an open file, it would result in 512 entire block reads from the same block.

## Functionality and Improvements

Our file system provides basic functionality for operating on files. It includes creating, reading, writing, and deleting files. The file system provides operations for performing both sequential and random access. The functionality is very minimal though and many common operations seen in other file systems are not supported.

One of the basic operations missing from the file system and the ThreadOS Kernel, is the ability to list the contents of a directory. Checking the existence of a file can be done using the SIZE system call, but seeing all files on the system cannot be done directly. This can be done indirectly by reading the content of the root directory, “/” using the OPEN and READ system calls. This is not a convenient API though and it leaks implementation details to the user of the system. Directory operations could be added to the Kernel and Syslib which could call operations on the FileSystem instance.

One of the core features missing from the file system is nested directories. One 1 root directory is supported in the file system. This would make supporting multiple users almost impossible and even for a single user, the single root directory would make managing multiple projects difficult as there could be few ways to organize files into groups and names could clash. Implementing nested directories is a potential improvement to the file system. The root directory itself is already stored as a file. An extra flag indicating that a file is a directory could help create nested directories.

Another core missing feature of the file system is the lack of metadata. Typical file systems store information about who created a file, when the file was created, and when it was last modified. This is helpful for multiuser systems and tracking changes on the system. This could be done on our file system by expanding attributes stored within an inode. This would mean that the space needed to store inodes would increase and IO operations would have to update metadata.

Another major missing feature from the file system is user access control. In a multiuser system, permissions are important. It would be beneficial to add improvements which allow or disallow specific users from reading or writing data on particular files. In order to implement such a feature, the concept of a user would have to be added to ThreadOS.

## Output and Testing

Testing of the file system was done using 2 methods. The first was using the provided test class Test5. This test checks the operation of the file system from a developer perspective. It has a few deficiencies though and an interactive tester was created to handle other cases.

## FSShell

The interactive tester is called FSShell. It can be run by loading FSShell. The file system shell will take input from the command line and try to match the input to a method in FSShell itself. It supports auto converting string parameters to booleans and integer values. This makes it easy to create new utility methods for testing.

### Test5 Output

The following output from Test5 shows all tests cases passing for the file system.

|  |
| --- |
| threadOS ver 1.0:  Type ? for help  threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)  -->l Test5  threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)  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-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  -->quit  Superblock synchronized |

### 

### FSShell Interactive Tests

#### Test: FileSystem Serialization and Deserialization

One of the tests not covered by the test case was checking that files are saved properly to disk on exit and can be read when ThreadOS is restarted.

The test starts out by:

1. Entering the fs shell
2. Formatting the disk for 5 max files
3. Creating 3 files
4. Existing the shell

|  |
| --- |
| threadOS ver 1.0:  threadOS: DISK created  WARNING: The disk is being auto formatted.  Type ? for help  threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)  -->format 5  format 5  -->l FSShell  l FSShell  threadOS: a new thread (thread=Thread[Thread-4,2,main] tid=1 pid=0)  fs> format 5  fs> write one.txt one  fs> write two.txt two  fs> write three.txt three  fs> dumpfs  fs [FileSystem@1741bae1] FileSystem  superBlock [SuperBlock@269c0b09] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [5] Integer  root [Directory@776e45ae] Directory  fsizes [[I@8184388] int[]  fsizes[0] [1] Integer  fsizes[1] [7] Integer  fsizes[2] [7] Integer  fsizes[3] [9] Integer  fsizes[4] [0] Integer  fnames [[[C@7666362e] char[][]  fnames[0] [/ ] char[]  fnames[1] [one.txt ] char[]  fnames[2] [two.txt ] char[]  fnames[3] [three.txt ] char[]  fnames[4] [ ] char[]  fileTable [FileTable@2f7a685e] FileTable  table [[]] Vector  dir [Directory@776e45ae] Directory  fs> exit  Exiting FSShell.  -->q  q |

In the next step:

1. Dump the details of the file system. It should show the file names properly. [PASS]
2. Read each file and check its content. [PASS]

|  |
| --- |
| C:\bryanc\education\CSS430-FinalProject>run  A subdirectory or file classes already exists.  "Compiling source....."  Note: Some input files use unchecked or unsafe operations.  Note: Recompile with -Xlint:unchecked for details.  threadOS ver 1.0:  Type ? for help  threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)  -->l FSShell  l FSShell  threadOS: a new thread (thread=Thread[Thread-4,2,main] tid=1 pid=0)  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [6] Integer  root [Directory@31715a25] Directory  fsizes [[I@1dd71fea] int[]  fsizes[0] [1] Integer  fsizes[1] [7] Integer  fsizes[2] [7] Integer  fsizes[3] [9] Integer  fsizes[4] [0] Integer  fnames [[[C@e24d466] char[][]  fnames[0] [/ ] char[]  fnames[1] [one.txt ] char[]  fnames[2] [two.txt ] char[]  fnames[3] [three.txt ] char[]  fnames[4] [ ] char[]  fileTable [FileTable@5cc1b06b] FileTable  table [[]] Vector  dir [Directory@31715a25] Directory  fs> read one.txt  Content:  one  fs> read two.txt  Content:  two  fs> read three.txt  Content:  three  fs> |

#### Test: Maximum File Name Size

Trying to create a file with more than 30 characters should show an error. [PASS]

|  |
| --- |
| fs> format 5  fs> write 123456789012345678901234567890BAD hi  Could not open: 123456789012345678901234567890BAD |

Writing a file with 30 characters should succeed and be readable. [PASS]

|  |
| --- |
| fs> write 123456789012345678901234567890 this\_has\_30  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [3] Integer  root [Directory@fc3f41d] Directory  fsizes [[I@697830db] int[]  fsizes[0] [1] Integer  fsizes[1] [30] Integer  fsizes[2] [0] Integer  fsizes[3] [0] Integer  fsizes[4] [0] Integer  fnames [[[C@386b9777] char[][]  fnames[0] [/ ] char[]  fnames[1] [123456789012345678901234567890] char[]  fnames[2] [ ] char[]  fnames[3] [ ] char[]  fnames[4] [ ] char[]  fileTable [FileTable@dfadc02] FileTable  table [[]] Vector  dir [Directory@fc3f41d] Directory  fs> read 123456789012345678901234567890  Content:  this\_has\_30 |

#### Test: Maximum Number of Files

The test starts by formatting the file system for 5 files. That should allow for the root directory and 4 extra files. 4 files are written to disk and checked that they can be read back. [PASS]

|  |
| --- |
| > format 5  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [2] Integer  root [Directory@4a6d5579] Directory  fsizes [[I@4e954bd9] int[]  fsizes[0] [1] Integer  fsizes[1] [0] Integer  fsizes[2] [0] Integer  fsizes[3] [0] Integer  fsizes[4] [0] Integer  fnames [[[C@5d7f1716] char[][]  fnames[0] [/ ] char[]  fnames[1] [ ] char[]  fnames[2] [ ] char[]  fnames[3] [ ] char[]  fnames[4] [ ] char[]  fileTable [FileTable@7538c7c9] FileTable  table [[]] Vector  dir [Directory@4a6d5579] Directory  fs> write 1.txt 1  fs> write 2.txt 2  fs> write 3.txt 3  fs> write 4.txt 4  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [6] Integer  root [Directory@4a6d5579] Directory  fsizes [[I@4e954bd9] int[]  fsizes[0] [1] Integer  fsizes[1] [5] Integer  fsizes[2] [5] Integer  fsizes[3] [5] Integer  fsizes[4] [5] Integer  fnames [[[C@5d7f1716] char[][]  fnames[0] [/ ] char[]  fnames[1] [1.txt ] char[]  fnames[2] [2.txt ] char[]  fnames[3] [3.txt ] char[]  fnames[4] [4.txt ] char[]  fileTable [FileTable@7538c7c9] FileTable  table [[]] Vector  dir [Directory@4a6d5579] Directory  fs> read 1.txt  Content:  1  fs> read 2.txt  Content:  2  fs> read 3.txt  Content:  3  fs> read 4.txt  Content:  4 |

The next attempt to create a new file should fail. [PASS]

|  |
| --- |
| fs> write 5.txt 5  Could not open: 5.txt |

Next deleting a file should leave an open slot. [PASS]

|  |
| --- |
| fs> delete 2.txt  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [6] Integer  root [Directory@4a6d5579] Directory  fsizes [[I@4e954bd9] int[]  fsizes[0] [1] Integer  fsizes[1] [5] Integer  fsizes[2] [0] Integer  fsizes[3] [5] Integer  fsizes[4] [5] Integer  fnames [[[C@5d7f1716] char[][]  fnames[0] [/ ] char[]  fnames[1] [1.txt ] char[]  fnames[2] [2.txt ] char[]  fnames[3] [3.txt ] char[]  fnames[4] [4.txt ] char[]  fileTable [FileTable@7538c7c9] FileTable  table [[]] Vector  dir [Directory@4a6d5579] Directory |

The next write should fill the opened slot and be readable. [PASS]

|  |
| --- |
| fs> write 5.txt 5  fs> dumpfs  fs [FileSystem@515ea56a] FileSystem  superBlock [SuperBlock@60c55592] SuperBlock  totalBlocks [1000] Integer  inodeBlocks [5] Integer  freeList [7] Integer  root [Directory@4a6d5579] Directory  fsizes [[I@4e954bd9] int[]  fsizes[0] [1] Integer  fsizes[1] [5] Integer  fsizes[2] [5] Integer  fsizes[3] [5] Integer  fsizes[4] [5] Integer  fnames [[[C@5d7f1716] char[][]  fnames[0] [/ ] char[]  fnames[1] [1.txt ] char[]  fnames[2] [5.txt ] char[]  fnames[3] [3.txt ] char[]  fnames[4] [4.txt ] char[]  fileTable [FileTable@7538c7c9] FileTable  table [[]] Vector  dir [Directory@4a6d5579] Directory  fs> read 5.txt  Content:  5 |

### Test: Attempt to Read Past The End

An attempt to read more data than is left from the current seek position to the end of the file, should result in reading only the remaining content. Read should return how many bytes read. [PASS]

|  |
| --- |
| fs> write test.txt 0123456789  fs> fopen test.txt r  Opened file: fd=3  fs> fread 3 9  Read: [9]  012345678  fs> fread 3 10  Read: [1]  9  fs> fclose 3 |