CSS430: File System

By Khuzaima Mushtaq

Introduction:

This document is a report on the File System project. We will cover the File System design and Implementation, Testing, and Functionality.

Design and Implementation:

My approach towards this project for design was a bottom up approach. Using the assignment guidelines, I understood the bigger picture of the file system but choose to focus on the components first and then build up from there. I used the outlines provided in the assignment description and notes to create the skeletal structure of the File System. I chose a bottom up approach because I think that designing independent modular components that interface together is a good approach to making an interface and to abstract the complexity of each component away from the others.

As required by the assignment my File System has the following components: Inode, Superblock, Directory, FileTableEntry, FileTable, and FileSystem with modifications to the Kernel to support the new system calls that we wish to emulate.

Inode:

The Inode is a file system component that keeps track of the nitty gritty details of a file in regards to a disk or hardrive. The inode is used to abstract the disk away from the rest of the file system such that everyone else sees the file as just a contiguous file. Whereas in reality the file could be in many different blocks in many different places on the disk. The Inode used in this project is a dumbed down version of the Linux Inode. Our Inode is 32bits long and all the data members of the Inode can be stored in 32 bytes and retrieved as such. Using the SysLib conversion methods provided it is trivial to convert and Inode object to 32bytes on disk and vice-versa.

My implementation of the Inode consists of a few more methods. addFreeBlock, createIndirectBlock, findBlock, freeIndirectBlock. These methods are defined as such:

```
    * adds free block to the inode as needed.
    * if direct blocks are open it uses those, otherwise uses indirect block
   */
  int addFreeBlock(int offset, short freeBlock){

   * createIndirectBlock
   * given blockNumber set up and write to disk the indirect block
   */
 int createIndirectBlock(short blockNumber){

   * Given an offset find a corresponding direct block
   * return the blockNum that it points to
   */
  int findBlock(int offset){

   * frees indirect block
   * and returns the data from the indirect block
   */
  byte[] freeIndirectBlock(){
```

These methods are the interface the Inode provides to the File Table so that the File Table is able to get and set freeblocks in order to read/write from an Inode.

Superblock:

The super block is the file system component that is responsible for maintaining an overall view of the number of total files, totalblocks available to the file system and for our implementation a freeList to keep track of free blocks that can be allocated. As required by the assignment the super block is the 0^{th} block on the Disk and can be read from or written to the disk. One important method of the superblock component is format.

```
 * Formats the disk and sets all variables to the proper values
 * Set up the FreeList
 * Write default Inodes to disk
 */
public void format (int numInodes){
```

format basically formats the disk such that we have space for numInodes number of files and sets up the FreeList for the rest of the disk

In addition to this, I added a few helper methods that are used by the File System: addToFreeList, and getFreeBlock and sync

```
 * Add blockNum to freeList
 */
public int addToFreeList(int blockNum) {

 * Retrieve a Free Block
 */
public int getFreeBlock() {

 * This method writes the Superblock to disk
 */
public void sync () {
```

These methods are used as described to make the job easier for the file system.

Directory:

The directory as the name suggests is the directory of the file system. The directory implementation of my file system is a single level directory. The basic structure is copied from the assignment and then implemented. The methods required for the directory are bytes2directory, directory2bytes, ialloc, namei. And ifree. I did not implement ifree as defined in the notes. I instead implemented a delete method that functions similarly but in the File Table. I chose to do this because the fileTable has access to the directory and can do the operation on both the directory and the FileTable. Ialloc is used to allocate a new filename to an iNode and namei is used to get the iNumber for an existing filename.

FileTableEntry:

The fileTableEntry provided in the notes for the assignment is complete and functional so I chose to use it as is rather than try to design a new one.

FileTable:

The file table contains a vector of FileTableEntires that is available to the file system. Users/ user threads can take the entries from the filetable and use them as file descriptors in their own file descriptor table. This is the access point through which users access files. The file table design is also based on the outline provided by the assignment. The fileTable also keeps track of the directory and updates it as needed.

The fileTable provides 4 methods: falloc, ffree, delete, and fempty. As mentioned above the delete is implemented in the fileTable rather than the Directory.

```
        * return the FileTableEntry related to the filename and mode
        * if the file already exists then simple point to it
        * otherwise create a new file with fileName and mode
        */
           public synchronized FileTableEntry falloc( String fileName, String mode )
{

     * remove fd/ FileTableEntry from the fileTable
     * update the inode and write it to disk
     * return success or failure
     */
           public synchronized boolean ffree( FileTableEntry fd ) {

     * find fileName in directory
     * use the iNumber to find the entry in the fileTable
     * delete/ free entry
     */
           public synchronized int delete( String fileName ) {

     * returns if fileTable is empty or not
     */
           public synchronized boolean fempty( ) {
```

File System:

The File System is the culmination of all the components of the file system. This is the interface through which user interact with files. The file table and the file descriptors are currency used in the file system to perform actions on the file system such as open, close, read, write, etc. The file system initialized and maintains the superblock the directory and the file table. The methods provided by the file system are: Sync, format, open, close, read, write, seek, delete, and fsize.

```
     * Synchronize the FileSystem and all its components to the Disk
     */
    public void sync() {

     * Format the Disk and the FileSystem and create a clean new FileSystem
     */
    public int format( int files) {
```

```
     * This method opens a file called filename with the permissions of mode
     * It returns a fileTableEntry which acts as a file descriptor that the users can
use
     * to interact with the file.
     * Creates a new file if filename doesnt exist, if it already exists just returns
the
     * FileTableEntry for it
     */
    public FileTableEntry open(String filename, String mode){

     * close fd
     */
    public boolean close(FileTableEntry fd){

     * Read from fd, into buffer
     * return value is the number of bytes read or -1 on fail
     * reads as many bytes as it can from fd into buffer
     */
    public int read(FileTableEntry fd, byte[] buffer) {
```

The implementation for read follows 3 steps:

- Using the seek pointer find the block associated to the seek pointer\
  - For example seekptr 513 = direct block 2 of Inode or seekptr 5000 = some indirectblock
  - For indirect blocks read the indirect block and parse it for the seek pointer and get the appropriate block
- Once we have the block, use the seek pointer to go to the exact byte in the block to start reading from.
- Based on the total length you can read vs blocksize and file length and buffer length
  - Test how many bytes to read from currentblock, for example total read = 600, since block size is 512 if we start at 0 we would only read 512 bytes from this block then move on to the next block and ONLY read 88 from the next block.
  - Update seek ptr and num of bytes read etc
- REPEAT until done

```
     * Write buffer to fd
     * Check mode to make sure writing is legal
     * returns num of bytes writter or -1 on fail
     * can write over data or append to file based on seek pointer
     */
    public int write(FileTableEntry fd, byte[] buffer){
```

The implementation of write follows the same steps as read EXCEPT we also have to add more blocks as needed to increase the file size and maintain the indirect block as well if needed. Otherwise follow the same steps.

```
     * seek the location in file based on whence + offset
     * adjust the file size accordingly and increment attributes as needed
     */
    public int seek(FileTableEntry fd, int offset, int whence){
      synchronized (fd) {
```

```
     * delete filename from FileTable
     */
      int delete(String filename) {

       * return file size of fd
       */
    public synchronized int fsize(FileTableEntry fd){
      synchronized(fd) {
```

Kernel:

As part of the assignment I also added the proper functionality to the appropriate kernel interrupts to execute the file system commands when called from SysLib as system calls.

Testing and Functionality:

This section covers the testing of the File System and the functionality

Test5:

```
threadOS ver 1.0:
threadOS: DISK created
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").....fd = 3 (wrong, css430 still exists!)
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
-->
```

These are the results of the Test5 provided by the instructor. The only failure is the file deletion which is because of my design. My attempt to implement delete in the File Table and not in the directory did not work out and the implementation is buggy. Otherwise the Test seems to work fine and the file system interface holds up.

My own testing of my file system was mainly focused on the components and ensuring the proper functionality of them. I used the SysLib commands improperly to only test component functionality outside of the file system. For example, Creating and maintaining Inodes, superblocks and testing the method functionality of each of their methods.

The file system as it is, is a very basic one, however it can be improved upon and made into a robust system. The directory structure can use much improvement and can be implemented as a multi level tree or acyclic graph. The Inode can be expanded upon to achieve the full functionality provided by the Linux Inode.