**Specification**

The purpose of this final project in CSS 430 Operating Systems is to implement a Unix-like file system using ThreadOS. User thread programs will no longer have to directly access disk blocks but instead be able to view stream-oriented files.

**Assumptions**

Our assumptions relied heavily on the specification given to us by Professor Fukuda and additional information from class lecture slides and notes. We also assumed the code already written in ThreadOS was sufficient for our needs and did not contain any bugs. We did not incorporate error handling for malicious users into our code because we assumed that every user would have knowledge of the file system implementation and not try to ‘break’ the code in any way.

**Limitations**

One limitation for our File System is that we only account for 64 inodes on our disk, which translates to handling 64 files at a time. For a real world operating system, we would expect it to be able to handle hundreds if not thousands of files at a time. For this reason, our implementation does not accurately reflect the scale or functionality of an actual real-world operating system.

Another limitation is there are not many layers of security between the different functions of our classes. Many of the methods and instance variables are public in each file and not encapsulated properly, which could limit the overall program’s functionality when scaling to bigger applications or limiting user’s access to various classes and functions. Methods that reference file system classes can use any function within the class, even ones that have more privileges, so this could be a limitation on the security of the file system.

**Internal design**

**Superblock.java**

The SuperBlock class reads the physical superblock from the disk and validates it to see if it’s usable.  It also provides a way of identifying free blocks, adding blocks to the freelist, and writing the new contents of SuperBlock back to the disk.  If validation fails, it will format, restore itself to an empty state and write a new SuperBlock to the disk.

**Inode.java**

The Inode class is used to describe one file. It stores 12 pointers of the index block, with 11 pointers to direct blocks and 1 pointer to an indirect block. Each inode stores the length of the file it describes, the number of file table entries that reference it, and a flag to store if it is used or unused. Each direct pointer contains the id of the index block to access its data directly, while the indirect pointer points to a block filled with short block ideas to directly access data. In total 16 Inodes can be stored in one block.

Our Inode class has two constructors, one to retrieve an Inode already stored from disk, and another to create a new empty Inode. When updating an Inode in memory, we check if the corresponding Inode has been updated on disk and read from it to memory and write back to the disk using the *toDisk* method. We also implemented methods to *registerIndexBlock, findTargetBlock, registerTargetBlock, and unregisterIndexBlock* for use in the file system when finding, allocating, and deallocating index blocks.

**Directory.java**

This class implements the Directory class in order to manage the files within the file system. Directory is maintained as the root directory and reads the file from the disk through inode 0 at 32 bytes of the disk block one, and stores this instance with the contents of this file. This class handles multiple files by constructing a 2-dimensional array to store each file’s name called *fnames* and a 1-dimensional array to hold each file size called *fsizes*. *Fsizes* is meant to store the sizes of each file in their allocated locations. *Fnames* stores the files within the Directory.

Upon instantiation by the File System, the Directory class takes a parameter of *maxInumber* to hold the max number of storage files to hold within *fsizes*. Then *fnames* gets initialized to contain the size and the maximum characters that the file name can contain. It then reads the file from disk from the inodes 0 to 32 in the disk block, and also writes back the file contents to the disk before shutting down. It also contains methods *bytes2directory()* and *directory2bytes()* to either read from the disk to a byte array and then write the byte array back onto the disk.

**FileSystem.java**

The FileSystem class handles the opening, closing, reading and writing of files.  When reading a file, it will rawread the contents of the file one block at a time and copy that into the buffer.  It will continue, moving the seekPtr and index as it goes until the buffer is full or the end of the file is reached.

When writing to a file, it follows similar logic. However, when writing it has to know if we’re trying to append or write to a new file. If the mode is set to write, the seek pointer (seekPtr) will find the specific location of the byte and begin writing to the file. If the mode is set to append, then we have to move the seekPtrto the end of the file, and instead of copying from the file to buffer, it’ll copy pieces from the buffer into a block and then rawwrite that block to the disk.

The *seek()* method updates the seekPtr to a corresponding file table entry and returns its location. The *deallocAllBlocks()* method takes a file table entry and checks all inode blocks, unregisters them, and overwrites their data with 0. The FileSystem also has a *close()* method that closes the corresponding file table entry and frees its entry within the file table.

**FileTable.java**

This class implements the FileTable class which maintains a set of file table entries within a Vector object. When a user thread opens a new file, its file descriptor number is stored within FileTable as reference to a FileTableEntry. The file entry is then allocated a new inode using the *falloc()* method given the filename and the access mode and recorded within FileTableEntry. The FileTable class also has a *ffree()* method that frees a file table entry from memory within the file table.

**FileTableEntry.java**

This class implements the FileTableEntry class which stores a reference to a file table entry’s corresponding inode and inode number. It also maintains a seek pointer variable to easily locate the corresponding file depending on the file access mode.

**Performance Estimation**

Our file system is currently adequate for the amount of information we are working with, but compared with real-world operating systems it has much to be desired. Its performance is not as fast compared to other high-level systems. This could be attributed to the fact that the disk blocks will be overwritten when deallocating blocks in order to protect file information as well as preventing files from being overwritten. We can also increase performance by allowing for more user multi-threads within the file system by clearing multiple blocks at a time and creating a new thread for every new deallocated block. However, implementing this would take much more work and might be out of the scope of this project for the time being.

**Current Functionality**

With our current implementation, ThreadOS is capable of formatting a disk block, opening, writing, and reading a few bytes from small files. It can also read large files with a size of approximately *512 x 13* bytes. The program also can delete appropriate files, create over 40 files, and correctly handle reading the same file with two independent file descriptors. However, currently our program cannot accurately append to large files or appropriately seek or write within large files containing *512 x 13* bytes.

**Possible Extensions**

Our file system could possibly be extended to accommodate for larger files by increasing the disk block size to hold more data. We could also increase the number of Inodes in order to read more than just 32 bytes of the disk block. Also, if we increase the integer value of *maxnumbers* in our Directory class, we could greatly increase the maximum number of storage files that it can hold within the directory and also increase the maxchars variable to hold longer filenames within the array of files. However, this might slow down the speed of indexing files and could result in a more costly runtime. With our current implementation, we ensure that while the amount of data is small, the filesystem is lightweight and relatively fast.