PROGRAM 5

Unix Style File System

Authors: Denali Cornwell & Jayden Stipek

Summary and discussions of algorithms, data structures, classes, and methods within Program 5.

Table of Contents

Compilation and Run Instructions

Compiled with:

- javac *.java
- This will give you:
    - Note: Kernel.java uses or overrides a deprecated API.
    - Note: Recompile with -Xlint:deprecation for details.
    - Note: Some input files use unchecked or unsafe operations.
    - Note: Recompile with -Xlint:unchecked for details.
- There is no need to worry about these notes, proceed!

Ran with:

1. java Boot
2. -->l Test5
3. You can do our test as well:
    a. -->l CustomFSTest

Assumptions and Limitations

We cannot find any assumptions and limitations based upon running Test5.java. We were able to run it multiple times on the same boot and it worked just fine. It seems that just using common sense and running the tests as they were intended did not produce any limitations to the software that we wrote for this project.

Performance Estimation, Current Functionality, and Extended Functionality

For current functionality we essentially have a fully fletched out basic filesystem. You are able to create new directories, read, write, close, open and the other necessities of a file system. Possibly for future extended functionality we could have it where you can rename a filename, instead of having to completely copy and paste the data from the files to do that currently. Another interesting task would be to make a GUI for a system like this.

Essentially make a Unix like GUI where you are allowed to do all these operations, but outside of a terminal and more customary, like an actual File system that's implemented with an OS. Of course these are goals for the future and would take time as all things do.

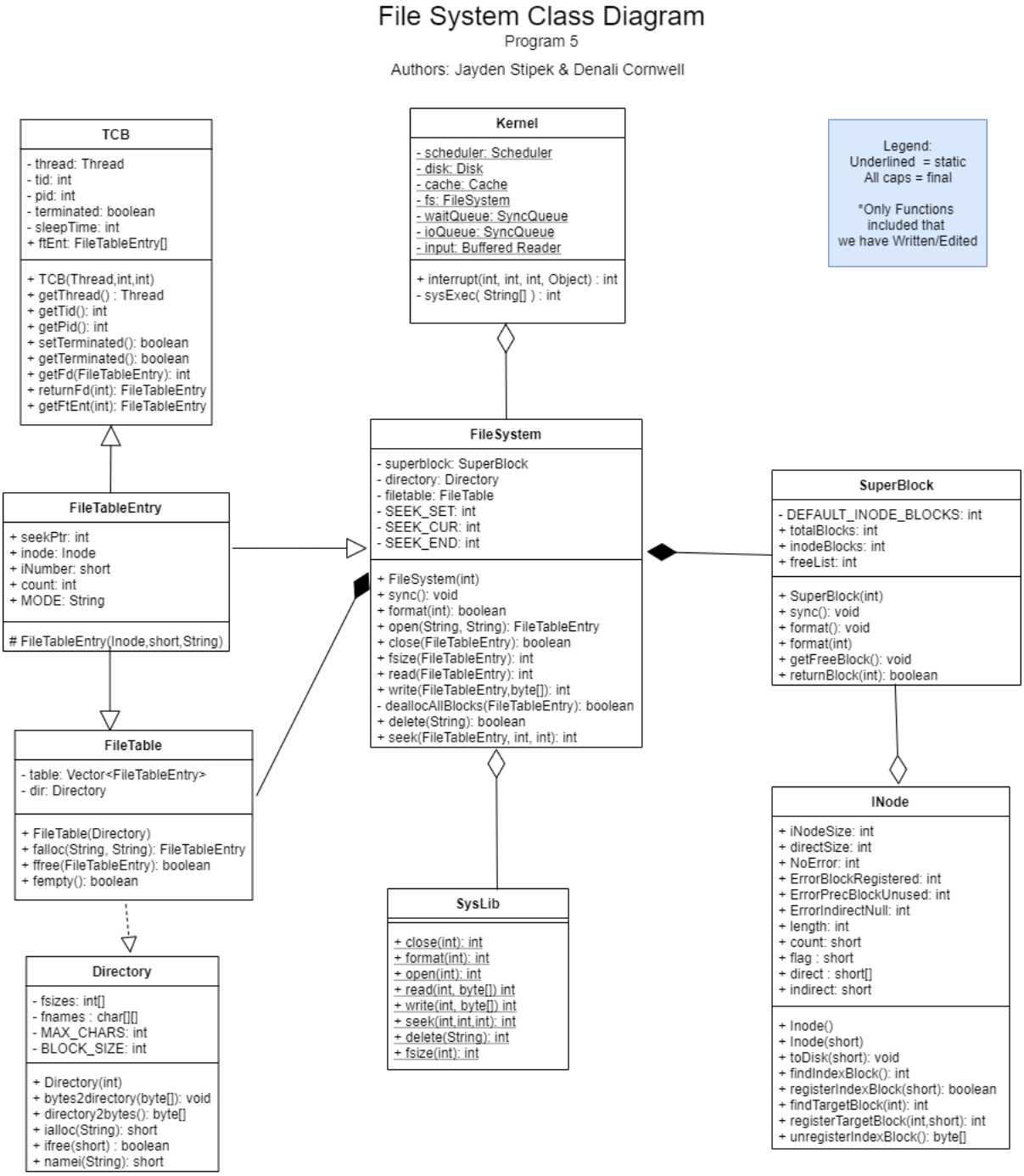
Output of Test5.java

```
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 )...................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
```

Output of CustomFSTest

```
threadOS ver 1.0:
Superblock synchronized
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l CustomFSTest
l CustomFSTest
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
Correct behavior of format(16).....................
Correct behavior of format(19).....................
Correct behavior of format(66).....................
Correct behavior of format(0).....................
Correct behavior of format(-1).....................
Correct behavior of format(1).....................
1: formating disk ( 64 )...................
Superblock synchronized
2: fd = 3-> 2: fd = 4-> 2: fd = 5    successfully completed
Correct sequence of fd on many openings.....................
3: Fsize = write( fd, buf[512] )....
Fsize check: successfully completed
Correct fsize on writing.......
4:Open for read and then write ..............
2: fd = 0 open( "css430", "r" )....
buf length 0Open for read and write: cannot write if the file is open for read
Correct: cannot write if the file is open in reading mode..............
-->
```

UML Class Diagram for Project



Discussion of Classes

This section delves into the purpose of each class that makes up the file system and discusses their methods and data members. We will only be going into detail about classes that we implemented.

SysLib.java

The SysLib class contains calls to interrupt the system through the Kernel.java class. This was the first class we modified to get the system working before doing any implementation beyond Kernel. All of the methods return the result of an interrupt call to kernel.

We added these functions:

int close(int fd)

Closes and fd from thread view. Using a call to the kernel interrupt function and returns the result from the call.

int format(int files)

Formats blocks for the file system. Using a call to the kernel interrupt function and returns the result from the call.

int open(String filename, String mode)

Open file for file system usage. Using a call to the kernel interrupt function and returns the result from the call.

int read(int fd, byte[] buf)

Reads from an file descriptor to a byte buffer. Using a call to the kernel interrupt function and returns the result from the call.

int write(int fd, byte[] buf)

Writes from a byte buffer to a file descriptor. Using a call to the kernel interrupt function and returns the result from the call.

int seek(int fd, int offset, int whence)

Moves the seek pointer across a file. Using a call to the kernel interrupt function and returns the result from the call.

int delete(String filename)

Deletes an frees a file in the file system. Using a call to the kernel interrupt function and returns the result from the call.

int fsize(int fd)

Gets the file size of a file at a descriptor. Using a call to the kernel interrupt function and returns the result from the call.

Kernel.java

The Kernel class handles all the interrupts from the ThreadOS system. The edits we made for supporting our new file system were to the interrupt method only. We added code to support interrupts for all of the functions we implemented in SysLib.java. Modification details are within the code, modifications primarily contain calls to the FileSystem object created in the BOOT case, and FileTableEntry's to be used as params for different calls to the FileSystem object.

SuperBlock.java

The SuperBlock class is the first low level class we completely implemented ourselves. We used the template on the Program5.pptx slides to get started.
The methods we implemented in SuperBlock:

void sync()
Synchronizes the superblock virtual memory to the disk using SysLib.rawwrite(int blockNumber, byte[] b). First the method creates a byte[] that then gets filled using SysLib.int2bytes(int I, byte[] b, int offset), where each integer data member that the superblock holds is converted to bytes, and placed into the byte[]. This byte array is then written to the disk with the call to rawwrite.

void format()
Simply uses the format(int nodes) function to format the standard amount of blocks supported by the superblock.

void format(int nodes)
Formats a certain amount of space for the file system. It creates a bunch of blank inodes, specified by the node's parameter. Then, it loops though each of those new nodes in the freelist and sets their block numbers using rawwirte.

int getFreeBlock()
Simply returns the int first free block in the freelist.

boolean returnBlock(int blockNumber)
Enqueues a given block at the front of the freelist. Creates a temp byte[] to be written to and given the free list integer, and sets the freelist integer to the block number indicating the block is now at the front of freelist.

Inode.java
Every file in the file system has an inode. An inode describes a file and hold all the metadata about the file so we can access in and know some key parameters. Inodes are used all throughout the file system implementation and control the direct and indirect blocks.

<u>The methods we implemented in Inode:</u>

void toDisk(short iNumber)
Writes the length, flag, and count to the disk, then writes the block number that the pointerIndex is pointing to. Once finished going through all the direct blocks, then reads the indirect block off the disk and then backs up all blocks with physical memory from the disk.

int findIndexBlock()
Simply returns the value for indirect as that is the pointer to the index block.

boolean registerIndexBlock(short indexBlock)
Allocates space for an indirect index block for the Inode. It makes sure to check that the block number input to the function does is not a direct block, if it is not then it will make a indexSetUp byte[] that will set up the index block so that it can use indirect pointers. If finally writes the bytes from the setup array to the indexBlock on the disk.

int findTargetBlock(int offset)
Returns an int that describes the target block specified in the index using an iNumber which is the offset passed in divided by the block size on the disk (512 bytes in our case). It will then use if-else logic to determine whether there is a valid block at the iNumbers location in the direct array, or search through the indirect index block if not.

int registerTargetBlock(int numBytes, int targetBlockNumber)
Selects a block and registers (allocates) it. This method does a similar equation as the last method to find the iNumber, but this time it is used to get a variable called offsetInt. If the offset is within the bounds of the direct blocks (< 11) then we can simply register a block there. If not, we create a data array for the indirect index block and allocates a block there. Returns the status of the target block.

byte[] unregisterIndexBlock()
Simply checks if the indirect block is present, if it is create an indexSetUp byte[] that will grab all the data from the indirect block and returns is, then sets the index block to -1 which means it is not in use. If the index block is already unvalid it will just return null.

Directory.java
Directory holds the actual names of the files within it. It manipulates them using a 2D array of chars fnames and each filename length in fsizes. Since fsizes holds each filenames length in its indexes, we can use it as a convenient way to find the index of a file within the directory.

The methods we implemented in Directory:

void bytes2directory(byte[] data)
First checks to make sure that the data being passed in is valid (is not empty, or null) then loops through the entire file sizes array using sysLib's (bytes2int variables) once done with that loops again through the entire fnames array to coordinate (sync) the fsizes and fnames arrays)

byte[] directory2bytes()
Creates a byte array the size of everything combined then does the loops through the fsizes array putting everything into the byte array. After that we loop through fnames changing everything first into a string and then copying that array into the "tempData" byte array and then finally returning the first byte array we created.

short ialloc(String filename)
Loops through the fsizes array and if any index is being used inside that array, then it allocates space for the name of the array.

boolean ifree(short iNumber)
You first check if the iNum is valid, if it is then you loop through the filename arrray and get rid of every char inside that array. Finally make sure you set the pointer to that index to 0 as well.

short namei(String filename)
Takes the filename as a string. Loops through the fsizes array and compares every index until you reach the filename it was looking for, or the end of the array in which it returns -1 if failed and the index if not.

FileTable.java
Keeps track of files and their statuses in a directory with a vector of FileTableEntry's. Can allocate space to open up a new file or free a space when deleting a file.

The methods we implemented in FileTable:

FileTableEntry falloc(String filename, String mode)

Uses filename and mode to allocate a new file in the system. First sets up an Inode that will be instantiated later depending on two cases, and an iNumber which will represent a block to allocate. It will use a while(true) loop and wait() statements to do busy waiting on threads so that no data gets mistakenly overwritten when trying to allocate the new file. First we want to set iNumber using dir.namei(filename) to see if a file is in the directory, if so, and they are in the correct modes we can break out of the loop and setup a new FileTableEntry using the inode, iNumber, and the mode passed into the function. Otherwise- if there is not a good iNumber, we should use the dir.ialloc(filename) method to allocate a new space for the file and create a blank inode and break out of the loop. In both cases, once broken out of the loop the function will return the newly allocated FileTableEntry.

boolean ffree(FileTableEntry e)

Tries to remove the entry from the table in the case of a delete. It decrements count on the FileTableEntry's inode.count and checks the flag on the inode to see if it is being read from or is in another condition. For the first condition it sets the flag in the inode to 0, and in the latter condition it will set the flag to 3. Finally, it writes the inode using the toDisk function of inode, and notifies any waiting thread that the process is complete, and returns true or false on failure or success.

boolean fempty()

Simply checks if the Vector of FileTableEntry's is empty. If so returns true, if not returns false.

FileSystem.java

FileSystem is the controlling class for all the classes we have implemented. It has a SuperBlock, FileTable, and Directory that will be controlled by the below methods.

The methods we implemented in FileSystem:

void sync()

Creates a FileTableEntry for the superblock. Creates a byte array for the directory, then writes that directory into physical memory. Finally closes and sync's the superblock.

boolean format(int files)

Does a busy wait until the file table is empty then formats the superblock using the given files. Afterwards recreates the directory and filletable based on the superblock information (inodeBlocks).

FileTableEntry open(String filename, String mode)

creates a FileTableEntry using falloc and the filename and mode. Then checks to make sure you are in the correct mode and if not returns null. Otherwise returns the created FileTableEntry.

boolean close(FileTableEntry ftEnt)

Makes sure you are the only one inside the method, then locks the binary integer lock so no one else gets in. Checks to see if anyone is currently using the FIleTableEntry that was passed in. If no one is using it then it frees the FileTableEntry and returns results, otherwise returns true.

int fsize(FileTableEntry ftEnt)

Synchronizes the method, and returns the length of the inode to get the file size.

int read(FileTableEntry ftEnt, byte[] buf)

First checks if the buffer is valid and the FtEntry is aswell. Makes sure you aren't in the incorrect mode either. Then finds the size of the buffer and synchronizes the method. While there is still information inside the buffer and your pointer isn't beyond the scope of the file size array: We find the target block index, then if it is not invalid (-1) we read that spot from the disk into a buffer array and then use that information to create a copy of that array for the virtual memory. Updating the pointer, the amount left in the buffer, and the data read throughout the process. Once finished with either of those you return the amount of data read in integer form.

int write(FileTableEntry ftEnt, byte[] buf)

Boolean deallocateAllBlocks(FileTableEntry ftEnt)

First checks to make sure that no one is currently using the FileTableEntry. After unregisters the index block and then does the same inside the superblock. Then precedes to get rid of the direct blocks and allocate them to -1 which means deallocated. Finally relays the information back to the disk which backs up the current data.

boolean delete(String filename)

Creates a FileTableEntry with mode write then if it closes the file and frees the directory properly, returns true, otherwise returns false.

int seek(FileTableEntry ftEnt, int offset, int whence)

Synchronizes the block to not allow anyone else in. Checks to see if they FileTableEntry is valid. Then finds what value was passed and based on the value determines where the seek pointer will be changed to. Finally returns the seek pointer.