Lance Fletcher, Jeremy Banks, Christopher Gong

Simple File System Design

Design

Our file system consists of an arrangement of 512-byte blocks that add up to 16MB in total. Our system is abstracted from a flat file, the structure of which is divided into three sections: Super region, Inode region, and Data region.
The Super region holds the metadata of the file system as a whole; specifically, the super region holds two bitmaps which denote whether a block in the flat file is free. The first bitmap is used to represent the state of inodes in the inode region, while the second bitmap denotes status of blocks in the data region.
The Inode region contains all metadata relevant to specific inodes. In our system structure, each inode gets its own block while the file system has 512 inodes in total. This is represented in the bitmap as char array of size 64 bytes. Each inode holds a stat struct which will maintain the metadata for the file it points to, as well as an array of 32 direct pointers to data blocks. Given this amount of pointers, each inode can reference a file of up to size $(32 \times 512) = 16$KB.
The Data region is where all of the contents of each file is written. Our file system holds 32,248 data blocks, meaning we have approximately 15.7MB available for files to write into. This is represented in the super bitmap as a char array of size 4031 bytes. As all relevant metadata is split between the super region and the inode region, the data region itself contains no metadata, only the bytes of files' contents.
The indices of inodes and files are determined by the bitmap. Each bit in a char byte in our arrays represents one block, either in Inode region or Data region. When an inode is created, its inode number will correspond to a spot in the bitmap. For example, looking through our char array, if the 3rd bit (index 2) of the 2nd byte (index 1) is free, that will correspond with the $((1 * 8) + 2) = 10$. It is important to note however that since our Super region is 4kb (8 blocks) in total, our Inode region will start at block index 8 as blocks 0-7 are reserved for the Super region. As such, all of our inode numbers will be greater than 8, with index 8 being reserved for our root inode. In the previous example the calculated block index would therefore be added to by 8, giving the index 18. As we have 512 inodes, the last inode will be at block index 519 while the first block for data will begin at index $(8 + 512) = 520$.

Files

Creation

To create (touch) a file, we initialize a new inode and fill in its stat struct. By default, all newly created files will be blank meaning no blocks will be allocated to this file, nor will the inode be initialized with direct pointers to any blocks. This will remain true until at least one byte is written into the file.

Reading

On a read (cat), we first get the number of bytes requested and compare that to the size of the file to be read. If (size request + offset) > size of file, we will return 0 bytes. Otherwise, we will record the number of blocks that belong to a file. Once we have that, we will go through that number of data blocks and for each block, read its contents and concatenate the result into one buffer. Once we have the entire file's contents as a string, we will copy the requested amount of bytes or as many bytes as possible into the return buffer. Lastly, we will return number of bytes read or set errno where appropriate.

Writing

The first part of writing is the same process as reading; we first load the entire contents of the file into a buffer. Once we have the whole file as a string, we will jump to the offset from the buffer start specified by the input parameter. At this point we will start writing in the string to file through the block_write function. Our string is split into chunks of 512 bytes to fill each block. Here, we have two cases: if the number of bytes requested is an exact multiple of 512, all blocks will be written to perfectly. Otherwise, we will fill as many 512 blocks as possible and for the final block, we calculate how many bytes to write, then fill in the rest of that block with null bytes.

Before returning the number of bytes that were written to the flat file, we update the size field of the inode whose file we just wrote to. To accomplish this, we compare the file's original size to the given write parameters: if offset + byte request > file size, the file will increase. The increase is calculated by finding the difference between (offset + byte request) and file size, and adding that on to the file size. For example, given a file sized 20 bytes and a request to write 10 bytes starting at offset 15 bytes, we would have (10 + 15) - 20 = 5 bytes. As such the size of the file after the write will increase by 5 bytes. Once the buffer was been written and the inode's size is increased appropriately, the inode is rewritten to its metadata section in the flat file and write returns the number of bytes read.

Removing

Deleting a specific file is fairly straightforward; rather than clear out a file's blocks on a delete call, we simply update the bitmap in the Super region to denote that file's blocks in Inode and Data region are available. Once this is accomplished, we can just remove that file's string from its parent directory's data section and consider it 'deleted'.

Directories

Data Region Structuring

Our directories are structured very similarly to regular files. Aside from setting the inode mode to denote a directory, there is no difference in metadata between a file inode and a directory inode. Where a directory differs from a regular file is in its data section; whereas a regular file's data section holds the contents of the file which are formatted in some way by the user, a directory's data contents are formatted in a very specific way: for each file in a directory, its data contents will be '[inode number] [tab] [filename] [newline]'. For example, each directory holds a reference to itself with the filename '.', so upon creation the data section of a directory would look like:

```
8	.
```

After a file is created in this new directory, we would see:

```
8	.
15	newfile.txt
```

Creation

As our directories are structured very similarly to regular files, not much will differ between file creation and folder creation. When a folder is created, we fill in all of its pertaining stat metadata. By default, all new directories will be given at least one block in data as we need to write in the current directory string.

Removal

Removing a directory is a more involved process than creating; when a directory is deleted, all entries inside that directory, either file or directory, must also be removed. To accomplish this, we recursively delete files inside a directory. First we traverse to the lowest level of any nested directories and flip the bits for every block of every file in that subdirectory. We then

File Searching

To find a specific file, we take its absolute path and parse that string to recursively jump through our file system. The way our directories are structured, each directory is a special file containing a file's inode number and corresponding file name. Given a path, we parse that path to get individual file names. Starting from root, we read root's contents to find a matching file/directory name. Once found, we grab that inode number to jump to the next directory. This process is recursively repeated, incrementing the path string to the next '/' character at each level. The search process is considered complete once we have a path containing no slashes. At this stage, if the current directory does not contain our target filename, we conclude that the target file does not exist. For example, given the path /folder1/folder2/tgt.txt our process would be:

- Get [folder1] from [root], use its inode number to jump to new directory.
    - New path is folder2/tgt.txt
- Get [folder2] from [folder1], use its inode number to jump to a new directory.
    - New path is tgt.txt
- Get [tgt.txt] from [folder2]
    - At this stage, no ‘/’ remains in the path meaning we have found our target

Implemented Functions

sfs_init

The init function is called the first time our file system is mounted. First, we will attempt to read the super region in our file system. If the super region doesn’t exist, we will initialize the file system with a new super. In this case, we will create a bitmap that denotes all inode and data blocks free except the first of each, which will be reserved for the root directory inode. The root inode is then initialized. By default, the root is given all permissions.
In the event that init is called while our file system already exists, init has nothing to do and thus immediately returns.

sfs_open

Our implementation of open serves only two purposes. First, open checks if a requested file path exists. This is accomplished through one of our helper methods, get_inode. This function takes in a directory path, parses it, and either returns the corresponding inode or sets a global flag if the file is not found. If a file is not found, open will return the errno ENOENT: no such file or directory.
The second purpose of open is to record the mode of the file about to be opened. A global mode field is maintained which holds the file type and permissions of whichever file was last opened. This field will be used during read and write operations.

sfs_create

The create function is where we will make all files in our file system. This is accomplished in three phases. First, we will search the inode bitmap in the super region to find a free inode. If a free inode cannot be found, create will return the errno ENOSPC: no space left on device. If an available inode is found, we will flip its corresponding bit using bitwise operations. Next, we will create the new inode, filling in all appropriate stat struct fields. When a file is first created it has no contents yet, meaning we initialize its size to 0 and do not provide it with any pointers to direct blocks. Once this inode struct exists in memory, we then write its contents to the inode region. Due to memory structuring, any structs we create may have some ‘padding’ bytes, so to avoid writing any junk data to the flat file, we convert the inode struct to a string and write that to the file using a write_to_file helper function. Lastly, we must update the directory which will hold this new file. In the open function where we call get_inode, as we traverse through directories to look for a file, we keep track of each file/folder’s parent inode. Once we find the parent node, we call our helper function writeToDirectory, which adds the newly created filename and its inode number to the data contents of the parent directory.

sfs_read

To read the contents of a given file, we check current permissions and use an auxiliary function named “get_buffer” that returns a char buffer from the data region. This function makes use of the given “block_read” function, and once we have successfully returned the char*, we “memcpy” it into the buffer provided and return a count of the number of bytes successfully read.

sfs_write

To write the contents of a given file, we check current permissions and use an auxiliary function named “loopWrite” that continuously writes a char buffer into the data region (which we memset everytime to clear out any possible added junk), 512 bytes at a time. This function makes use of the given “block_write” function, and once we have successfully wrote the buffer that was passed, we return a count of the number of bytes successfully wrote. If the file expands after a write such that its size exceeds the 512 byte-limit for a block, we make use of another one of our inode direct pointers. In the event that we need to obtain more blocks to write into, we will again traverse our bitmap to look for a new free block. If a write operation cannot be fully completed due to a lack of block availability, we write in as many bytes to the file as possible and return ENOSPC (not enough space available).

sfs_unlink

To delete a file, we have two steps. First, we ‘remove’ the file by flipping the bits of the file’s inode and data blocks in the bitmap of the Super region. Since we clear out data blocks during a write operation, we don’t care about the bytes still remaining in a data block but rather leave them as ‘junk data’. Once all the file’s bits are reset, the only other step is to remove the file name from its parent directory. To accomplish this, we call our writeToDirectory function with a delete flag.

sfs_getattr

When sfs_init initialized our file system’s main data structure, sfs_getattr retrieves data from it. Essentially, this comes down to simply calling our “get_inode” function, while passing root’s inode as a parameter, which is kept as a global. Since our inode struct contains the required stat struct within it, we simply assign the stat struct values within our retrieved inode into the statbuff parameter.

sfs_release

Release (to our understanding) serves the purpose of freeing or clearing any structures or fields we maintain while a file is open. However, we do not allocate any objects for maintenance while a file is open and as such, we have nothing to free up when a file is closed. Our release therefore does not do anything.

sfs_destroy

Sfs_destroy is called when our directory is unmounted. At this point, our only operation is to call disk_close. We don’t clear out our flat file because we chose to have our data persist after unmounting in the event that the user would like to mount again at a later time to access the same files created at an earlier point.

sfs_readdir

Sfs_readdir grabs an inode based upon a passed path, which we then call “get_buffer” upon to return a char* containing every file and sub-directory within that directory. We then continuously parse said char* by delimiters that we passed, placing each into an empty stat struct while using the given filler() function.

sfs_opendir

Opendir’s function is to check whether a directory exists. We call get_inode to search for the requested directory and check whether the function sets the fileFound flag. If the file is not found, opendir returns ENOENT (no such file/directory). Otherwise opendir records the directory’s mode and passes it on.

sfs_mkdir

As our implementation of directories differs very little from our file implementation, there are minor differences between mkdir and create. First we search the Super region’s bitmap to find a free inode and a free data block, returning ENOSPC if either of these fail. Accomplishing this, we then call writeToDirectory first to the parent directory, then we call writeToDirectory on the just-created directory, adding its own filename and inode number.

sfs_rmdir

Similar to our implementation of sfs_unlink, sfs_rmdir removes a single sub-directory from a given directory. The main difference being that we need to also remove all files and nested directories within that directory that we want to delete. To do this, we created an auxiliary function named "removeSubDir" that recursively deletes all entries contained in the directory. Inside the function is a loop that checks every entry within the directory; if it's a directory, we recurse, otherwise sfs_unlink is called on that file.

sfs_releasedir

Same as sfs_release, since we do not allocate any global structs or fields for open directories. As such, we have nothing to do in releasedir.

Testing

To test this program, we started small, but the tests themselves exceeded in granularity as we went:

1. To make sure the initial functions were in working order, we performed small individual tests that would simply call sfs_init() and sfs_getattr(). Doing this early on allowed us to assure that our inode data structure was able to be created [sfs_init()] and grabbed [sfs_getattr()] properly, since these functions are essential to implement before any of the actual operational functions.

2. From there, we knew that the next step would be to test our functionality of creating actual files in our file system. After implementing sfs_open, sfs_create, and sfs_readdir, we simply "touched" multiple files, followed by an ls command. Once we saw that multiple files were able to persist in the flat file, we knew we could proceed.

3. Once we knew that that we could accurately touch files into our file system, we implemented the sfs_unlink function. To test this, we simply touched a collection of three files and removed all of them individually and in an arbitrary order. This action was repeated until it was clear that we could remove files in any order.

4. Once we knew that we could touch and rm files, it was time to implement reading and writing. To test this, we simply touched a file, performed an "echo" command, and then

performed a “cat” on the same file. Once we could reliably read the same thing that we “echo”-ed with a “cat,” it was time to test if we could write/read between multiple blocks within our file system. To do this, we “echo”-ed a string into a touched file that was greater than 512 bytes (the size of a block), and once we could accurately “cat” that same file, we knew that we could correctly store the data between blocks/different direct pointers.

5. Since we decided to implement two of the extra credit segments, our next major test was for the functions sfs_opendir, sfs_releasedir, sfs_rmdir, and sfs_mkdir. For this portion, we simply performed a “mkdir” command to create a new directory, and once we could do that and “cd” into it, we conducted a further series of mkdirs and touches that assured us that we could create nested files and directories. Once our tests achieved that, it was time to check the accuracy of a “rmdir” command. To test this, we created a sequence of nested directories that all had exactly one regular file within them. Then, we performed a “rmdir” command on the top-most directory that we created, checking to make sure that all sub-directories and files were correctly removed.

6. At this point, all of the major functions were in place, so it was time for a single huge series of tests to assure that our file system was in working order. To do this, we created a single shell script to automate our series of commands. To know that our root inode would be using multiple direct pointers, we touched 128 files with the name “`heyThisIsAReallyLongTestFileNameAgain[number].txt`”, which assured us that the root inode string that we created would exceed the block-limit of 512 bytes. Then, one-by-one, we “echo”ed and “cat”ed every file that we just touched with a string that exceeded 512 bytes themselves. Once this very large test consistently succeeded, we knew that our file system could hold hundreds of files that all contained data that requires multiple direct pointers to both read and write, proving to us that our file system remained strong despite such a large test.
7. For a final test, we tried writing from one file directly to another. We wrote about two blocks worth of data into one file, then cat-ed that file into another file. This appeared to consistently work*. (* see error assumptions)

Extra Credit

In addition to doing the base requirements of the project, our team also went the extra mile and implemented both *Directory Support* and *Extended Directory Operations*. These additions were described in detail above in these function-specific sections: sfs_opendir, sfs_releasedir, sfs_mkdir, and sfs_rmdir.

Error Assumptions

- While our program creates the flat file if one does not exist, we do not check for the pre-existence of a mount directory. While the mount directory's location does not matter, it may prove beneficial to use the directory: '/tmp/laf224/mountdir'. (see testing instructions for details)
- When we attempt to cat from one file to another the terminal prints a 'bad address' error. Such a call would look like this: "cat file1.txt >> file2.txt". Despite this error, we can still read from and write to the target normally. As such, we are assuming this to be a non-issue.
- Due to a built-in preset, fuse will only write a maximum of 4kb to a file in a single write operation. As such we assume that no write operations inserting a buffer larger than 4kb at once will be performed.

Testing Instructions

- We have included a header file, 'sfs.h', and have updated the makefile to reflect this. To make things simpler for everyone, we simply submitted our entire modified src directory. To compile our program, please use the makefile included in the src folder.
- As an alternate method of compiling, we have included a script to automatically setup our file system. By going into the src folder in terminal, our script can be called using the command: '. ./setup'. This command will automatically create all necessary files. Additionally, if the command is called with a '-c' (ex: . ./setup -c) flag, the terminal will automatically jump to the mount directory.
- To unmount our file system, we have included another command. In the src folder, the command './unmount' can be used to automatically unmount our file system and delete all related files. To keep specific files after unmounting, include the -s flag and list the names of all directories to be kept. (ex. To unmount but keep the flat file, use: ./unmount -s testfsfile)