Group Members: Daniel Parks, Eric Knorr, Eric Fiore

Basic Structs and Metadata

In order to represent files and directories in our file system, we make use of only three structs (not including the struct sfs_state): inode_t, path_map_t, and file_entry_t. These structs would be the foundation for three tables that are foundational to every file operation in our assignment. A table of inodes would be the primary representation of actual data in our SFS, with the array index representing the inode number, and each entry containing a stat structure, and a table of block numbers to indicate where the data was stored on the disk file. For smaller files, we designed the SFS to work with indexing, to speed up access. However, for very large files, we implemented double-layer indirect indexing to reduced the space taken up in the inode table, while maintaining fast access times for small files (see "Indirect Indexing" header). The second of the three main tables in our SFS is the name table. This structure maps absolute file paths to inodes in the inode table. We designed our SFS with the following philosophy in mind: file names and data should be stored separately, Linux-style. The third and final table is the open file table. This table represents the files that are currently open, and is our way of representing file handles in our SFS. We used the fi->fh member in the fuse_file_info struct to store a file handle that is actually the array index of an entry in the open file table. If a file handle is passed as an argument, the program will bypass the name table, and be able to access the inode table directly, making file operations more efficient. In addition to these three struct tables, there is one final table that is simply a bitmap indicating what disk blocks are available. The indexes of this array are the values of entries in a given inode's block table.

Persistent Storage, sfs_init(), and sfs_destroy()

Our file system is 100% persistent. When our SFS is first mounted, it calculates the space taken up by all tables and metadata, and reserves space at the end of the disk file for storage upon unmounting. When the user unmounts our SFS, sfs_destroy() will write a 64-bit integer key (randomly generated and defined as a macro) to the beginning of the first system-reserved disk block. Then, the SFS will write all tables and metadata to the remaining reserved disk blocks.

When the SFS is mounted, sfs_init() will start with the last disk block and move backwards, searching the number of blocks that would be required for system metadata, attempting to match the 64-bit key. If sfs_init() finds a match for the key, the metadata will be loaded into static memory from the file, making all previous operations persistent, assuming the file has not been changed in between mounts. If a key is not found, all metadata and tables are initialized to default values, and the root directory is manually entered into the tables as a regular entry.

Obtaining Metadata and sfs_getattr()

Being the most important internal operation to our SFS, we made this function as simple and effective as possible. When called, the SFS will perform a pathname lookup. If the file is found, the function will simply copy the stat structure from the corresponding inode into the caller-supplied buffer. If the file is not found, a stat structure initialized to default values for a regular file will be returned.

Regular File Operations

Most file operations take the following form: take an absolute path, perform a path name lookup, access the corresponding inodes, perform a calculation and a corresponding operation. The functions that operation on regular files (not directories) are summarized here:

sfs_create(): this function creates and opens a regular file. After checking that the file does not already exist, it will search for a free entry in the inode table, the name table, and the open file table. If any of these allocations fail, the function will return an error. After the metadata is allocated, the current working directory is inferred from the path variable, and the new file name will be entered into that directory's data (see "Extended Directory Operations" for more on this). Newly created files are *not* allocated blocks by default, and begin with zero size. Additional blocks are allocated to the directory inode as needed to accommodate new data on a first available (lowest block index) basis.

sfs_unlink(): this function works as the exact opposite of sfs_create(). After checking the arguments, it deallocates entries in all relevant tables/metadata.

sfs_open(): this function allocates an entry in the open file table, and assigns a file handle to the caller-specified path.

sfs_release(): this function operates as the exact opposite of sfs_open(), and deallocates the entry in the open file table corresponding to the caller-supplied path.

sfs_read(): this function performs a lookup based on the file handle, and reads data off of the disk. Because disk operations can only be done by the block, the function will allocate a new buffer in addition to the caller-supplied read buffer that is aligned with BLOCK_SIZE. The function then calculates how many blocks it will need to read for the operation, loads them into the continuous, aligned buffer, and then transfers the data from the aligned buffer to the caller-supplied buffer.

sfs_write(): this function operates nearly identically to sfs_read() and makes use of an aligned, continuous buffer in addition to the caller-supplied buffer. The relevant disk blocks are loaded into the aligned buffer, the data is then copied from the caller-supplied buffer to the aligned buffer, and then blocks are then written back to the disk. New blocks are allocated to files as necessary to accommodate new data on a first available (lowest block index) basis.

Extended Directory Operations (extra credit)

In our SFS, directories are nothing but files that contain special data. They are mapped to the meta tables in a way identical to normal files. The data in a directory takes the following form: "./<inode*>/../<inode**>/<entry>/<inode>/ …" where inode* is the inode number mapped to the directory itself and inode** is the inode mapped to the parent directory. All directory operations deal with this data in some way.

sfs_readdir(): this function will perform a pathname lookup on the caller-supplied path, read the directory's data off the disk, and tokenize the entries using "/" as a delimiter. The entries will then be passed to the call-supplied buffer with the filler() function.

sfs_opendir(): we left this function unimplemented, as it was unnecessary for our needs.

sfs_releasedir(): we left this function unimplemented, as it was unnecessary for our needs.

sfs_mkdir(): this function operates nearly identically to sfs_create(). It performs all of the same operations on metadata as sfs_create(), except it allocates one disk block to the new directory and writes the default entries ("./<inode>/../<parent inode>") to that block. The function also infers the current working directory from the caller-supplied path, and appends the new entry to that directory's data to reflect the new entry.

sfs_rmdir(): this function works almost exactly the same as sfs_unlink() with a few differences. The function will scan the directory's data to see if there are any entries that are not default, and returns an error if there are. If not, the function will deallocate all relevant resources and remove the entry from its parent directory.

Indirect Indexing (extra credit)

In our file system, each inode contains a list of block indices denoting the blocks in which the given file or directory's data is stored. Each inode/file may have up to 1110 blocks or 568320 bytes of data. In practice, each inode contains an array of 12 block indices. The first 10 of these indices point to the first 10 blocks of data. The 11th and 12th indices each point to blocks which contain additional indices, the 11th pointing to a block containing the indices for the next 100 blocks of data and the 12th pointing to a block containing 10 more block indices, each of which in turn point to blocks containing the indices for another 100 blocks of data. Therefore, files with 10 or fewer blocks will require a single disk access per block access, files with 110 or fewer blocks may require up to 2 disk accesses per block access and files with 1110 blocks or fewer may require up to 3 disk accesses per block access. Whenever a new block is allocated to a file containing more than 10 blocks, the new index is immediately written to disk in the respective block, so all file indexing is persistent.

Testing

Initial testing consisted largely of running simple commands such as ls, touch, echo, cat and rm and following their execution by attaching gdb as well as unmounting and remounting to check for persistence. For further testing we used scripts to write out large files (to test block allocation and indirect indexing), create large numbers of small files (to verify inode capacity and block allocation for directories), make trees of nested directories with files (to test extended directory operations) as well as create files, remove some and then create more (to verify proper allocation, deallocation and reallocation of blocks). We could then browse through the files and directories as well as examine the data written to disk to verify the operations were successful.