CS518: OS Design
Assignment 3: FUSE file system

Members

Roshan Patel	rrp78
Daniel Hidalgo	dh624
Saad Ismail	sni7

Build Instructions

Our file-system implementation is already mounted on our group's assigned ilab virtual machine: `classvm24.cs.rutgers.edu`. We have been accessing this VM as the user 'csuser'.

The directory of our mounted file system is:
`/home/csuser/submission/example/mountdir`

The location of the file system's flat file is:
`/home/csuser/submission/optimusprime`

To build our sfs executable to create and mount our file system from scratch, follow this procedure:

1. Copy provided /src into the provided FUSE directory.
   ```
   > scp -r /src → classvm24.cs.rutgers.edu:~/fuse-dir/
   ```

2. Configure and make
   ```
   > cd ~/fuse-dir/
   > ./configure
   > make
   ```

3. Create a new file-system flat file
   ```
   > touch megatron
   ```

4. Execute and mount file-system
   ```
   > cd example/
   > ../src/sfs ../megatron mountdir/
   ```

Files Submitted

Our submission on sakai includes all of the files within `/src` plus this readme.

block.c	Block IO implementations
block.h	Block header file
config.h	Provided, unaltered.
copy_src_over.sh	Bash script for scp
fuse.h	Provided, unaltered
inode.c	Inode struct implementation
inode.h	Inode header file
log.c	Provided, unaltered.
log.h	Provided, unaltered.
Makefile	Provided, unaltered.
Makefile.am	Provided, unaltered.
Makefile.in	Provided, unaltered.
params.h	Sfs_state struct definition
README.md	Dev-notes. Not this readme. Ignore.
sfs.c	Sfs-functions implementations
stamp-h1	Provided, unaltered.
list.h	Helper functions for circular LL's.

Accomplishments

Our file-system supports one working directory (root directory “/”) and up to 7 files within it. Each file is designed to support up to 16850944 bytes (~16MB). Our implementation is able to support more files by changing an inode-count parameter, but we intentionally kept the number small as to reduce the load on the VM during testing.

Using our file-system, one is successfully able to create, delete, open, read, and write files. For example, the following bash commands operate as expected:

```
> cd mountdir/
> touch autobots.txt
> echo “bumblebee” >> autobots.txt
> cat autobots.txt
> ls -la
> rm autobots.txt
```

Note that the ‘touch’ command returns this message:
```
“touch: setting times of ‘autobots.txt’: Function not implemented”
```
Despite this message, the command operates as it should. A file is created if it already doesn’t exist, and a file’s time accessed is properly updated if it already exists. We have not figured out how to remove this message or trigger it to not print.

Indirection pointers are implemented and function in our file-system. File sizes of up to 73728 bytes can be written (144 block sizes). We have implemented test scripts that help exercise this. (see Testing section).

```
> cd ~/submission/examples/
> sh test04_oneindir.sh                  # writes to 17 data blocks
> sh test05_moreindir.sh                 # writes to 20 data blocks
> sh test06_fullindir.sh                 # writes to 144 data blocks
```

Although the infrastructure for double indirection is in place, we were unable to fully implement writing and reading files using the double indirection pointers. Thus the max file size is 73728 bytes.

Design Overview

Our file-system is composed of three sections: a superblock section, an inode section, and a data section.

The superblock section contains important metadata regarding the indexes of the inode and data sections, details about the current state of the file-system, and a secret number to check the validity of our file-system's initialization.

The inode section contains all of the file-system's inodes. The FS uses inodes to keep track of a file and its data blocks. We implemented our own inode struct with inspiration from the original UNIX file-system's implementation. It is located within inode.c and inode.h. Each inode contains metadata, 16 direct mapped pointers, 1 indirect mapped pointer, and 2 double indirect mapped pointers. The size of all inodes is fixed at one-quarter of one block (inode = 128, blocksize=512). A direct pointer will point to one data block. An indirect pointer will point to an entire block (512 bytes) containing 4-byte pointers (128 pointers) to data blocks (i.e. 1 indirect = 128 data blocks). A double indirect pointer will point to an entire block of indirect pointers (i.e. 1 double indirect = 128 indirects = 128*128 data blocks).

The data section contains all of the blocks reserved for user data and some file-system functions. The root directory utilizes inode 0 and has data blocks reserved for it to store directory-entries. Indirection and double indirection pointers utilize pnodes. Pnodes are block-sized structures that contain 128 pointers to data blocks. These pnodes are also located within the data section.

On file-system initialization, the superblock, inode section, and data block section are allocated using block IO calls (see block.c/block.h). While creating data blocks, each allocated data block is initially pointed to by an inode. Thus, at the end of sfs_init(), all of the inodes will have valid pointers that resolve to data blocks for direct, indirect, and double indirection (pnodes are also initialized). Strict control is kept over which data blocks are currently in use and the next available data block pointer within the inode's meta data. When the file reduces or increases in size, the inode will respond to the update properly.

Free space is managed using a free linked list. Data blocks are all pre-allocated to inodes, so inodes internally manage their own free space. Locating a free inode is done using a stack structure implemented as a circular linked list. Upon initialization, all the available inode's ino (inode numbers) are enqueued. As a new file is created, this value is dequeued and used for the new file. As a file is deleted, it's inode ino is enqueued back into the list. For added security, each inode contains a validity field.

Relevant Structures

```
typedef struct __attribute__((packed)){
    uint32_t    isvalid;                    // IS THIS INODE VALID?
    uint32_t    ino;                        // inode number
    uint32_t    mode;                       // type of inode: dir/file/direct
    uint32_t    size;                       // total size of dir/file in bytes
    uint32_t    num_blocks;                 // total number of blocks
    uint32_t    nlink;                      // number of hard links
    uint32_t    current_unused_indx;        // index of blocks array thats is currently unused
    uint32_t    current_unused_bno;         // blockno of next unused block
    uint32_t    time_access;
    uint32_t    time_mod;
    uint32_t    time_change;

    uint32_t    blocks[SFS_TOTAL_PTRS];     // stores blocknos of direct, indir, and dindir
}sfs_inode_t;
```

The above is our inode struct declaration. It contains metadata about the file it is associated with along with an array called blocks. The metadata includes most of the generic information that the UNIX FS uses for its inodes. In addition, we have added a valid bit and two ints indicating which blocks/indexes are currently unused by the file. The blocks array is an array of 19 ints. 16 are direct mapped pointers, 1 is an indirect pointer, and 2 are double indirect pointers.

```
typedef struct __attribute__((packed)) {
    uint32_t    ino;                                // inode number
    char        name[SFS_MAX_FILE_NAME_LENGTH];     // file name
}sfs_direntry_t;
```

The above is the direntry struct declaration. The direntry struct servers to map an inode's inode number to its String filename. There is one direntry for each file that exists. A directory inode will store these direntrys in data blocks that it points to. The max file name length is 32 characters.

```
struct sfs_state {
    FILE *logfile;
    char *diskfile;
    list_node_t* free_inodes;       // List of free inos (connected via sfs_item->node->next)
    uint32_t root_ino;              // ino of root dir.  "/"
};
```

The above is the sfs_state struct. We have altered it by storing the root's inode number and a pointer to the head of the free inode list.

Implementation

In order to satisfy the requirements of this project, we needed to implement all of the functions in sfs.c. The bulk of data structure management is handled in inode.c and sfs.c calls helper functions from inode.c to accomplish tasks.

The following are the helper functions defined in inode.c:

```
uint32_t ino_from_path(const char *path);
uint32_t ino_from_path_dir(const char *path, uint32_t ino_parent);

void update_inode_data(uint32_t ino, sfs_inode_t *inode);
void update_block_data(uint32_t bno, char* buffer);

uint32_t get_new_ino();

void create_direntry(const char *name, sfs_inode_t *inode, uint32_t ino_parent);
void read_direntries(sfs_inode_t *inode_data, sfs_direntry_t* dentries);
void read_direntry_block(uint32_t block_id, sfs_direntry_t* dentries, int num_entries);
void remove_direntry(sfs_inode_t *inode, uint32_t ino_parent);


uint32_t create_inode(const char* path, mode_t mode);
int remove_inode(const char *path);
void populate_stat(const sfs_inode_t* inode, struct stat *statbuf);
void get_inode(uint32_t ino, sfs_inode_t* rtn_inode);
int read_inode(sfs_inode_t* inode, char* buffer, int size, int offset);
int write_inode(sfs_inode_t *inode_data, const char* buffer, int size, int offset);
```

The respective ino_from_path functions resolve an inode number from a String path utilizing direntries. There two functions in preparation of handling a recursive folder structure, but we did not have time to implement multiple directories.

The update_*_data functions use a buffer to copy entire blocks to the file system flat file using functions from block.c.

There are a set of functions that handle navigating through direntries, creating direntries, and removing direntries. All of these functions help enable bash commands like ls and rm.

The last chunk of helper functions assist with inode management. They implement the bulk of file creation, writing, reading, and removing.

SFS_Functions

1.`void *sfs_init(struct fuse_conn_info *conn)`

This function checks the disk file if it is already initialized by verifying the size of the file system in the super block. If the size is not zero, it verifies the magic number if it is a valid disk file that can be read by our system. If both those conditions hold, init returns after gathering the metadata from the super block(inode location, max file size, etc.), otherwise we initialize the file system if the size is zero. Using the helper functions from above, the max number of inodes are allocated in contiguous disk blocks along with a list of free blocks. If the size was nonzero and the magic number did not match, init returns an error message in the log saying it is not a valid file system.

2.`void sfs_destroy(void *userdata)`

This function just closes the disk_file using the function in block.c.

3.`int sfs_getattr(const char *path, struct stat *statbuf)`

This function first converts the given path name to an inode number in the file system. If no inode number is found, ENOENT is set and returned, otherwise the given stat struct is populated with values from our inode struct.

4.`int sfs_create(const char *path, mode_t mode, struct fuse_file_info *fi)`

This function calls our helper function create_inode which looks for an available inode in the file system to initialize.

5.`int sfs_unlink(const char *path)`

This function calls our helper function remove_inode which clears the inode, frees any data associated with it and then makes it available to be reused.

6.`int sfs_open(const char *path, struct fuse_file_info *fi)`

This function verifies if the given path name resolves to an inode. Once an inode is retrieved , its mode is checked if it can be opened.

7.`int sfs_release(const char *path, struct fuse_file_info *fi)`

This function retrieves the inode for the given path name and modifies its time_access field to indicate it is being closed.

8.`int sfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`

This function makes a call to our helper function read_inode which find the first block to read from and iterates through our direct and indirect blocks (if necessary) to store the data into the given buffer.

9.`int sfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`

This function is similar to read except performs the write operation instead and calls the write_inode helper function.

10.`int sfs_mkdir(const char *path, mode_t mode)`

This function calls the helper function create_inode which is similar to sfs_create but the mode will be different in this case to distinguish between the inodes.

11. `int sfs_rmdir(const char *path)`

    Not implemented

12. `int sfs_opendir(const char *path, struct fuse_file_info *fi)`

    This function calls the helper function get_inode which retrieves the inode from the given path name. The inode is then checked to see if the mode matches S_ISDIR.

13. `int sfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,struct fuse_file_info *fi)`

    This function calls our helper function read_direntries which populates the entries in the directory into an array.

14. `int sfs_releasedir(const char *path, struct fuse_file_info *fi)`

    Not implemented

Testing

We've tested the functionality of our file-system with certain test cases. The scripts are located in the following directory:

```
/home/csuser/submission/examples/
```

1. `test00_multifile.sh`
   Tests creating multiple files. Exercises the creating new inodes using the free list.

2. `test01_simpleonefile.sh`
   Tests creating, writing, reading, and deleting one file.

3. `test02_clone.sh`
   Creates a file and writes multiple blocks work of data into it.

4. `test03_justunder.sh`
   Creates a file and writes enough data to fill all 16 direct blocks. (8192 bytes)

5. `Test04_oneindir.sh`
   Creates a file and writes 17 blocks worth of data (8704 bytes). This ensures that at least one block points to a data block pointed to by an indirect pointer. The proper execution of this test ensures that the base of indirect pointers are functional.

6. `Test05_moreindir.sh`
   Creates a file and writes 20 blocks worth of data (10240 bytes). This test stresses the use of data blocks pointed to by indirection pointers.

7. `test06_fullindir.sh`
   Creates a file and writes 144 blocks worth of data (73728 bytes). This test completely saturates all the data blocks points to by both the direct and indirect pointers.