CS416 Project 4: Tiny File System using FUSE Library

Due: 05/03/2019

Points: 100

In Project3, you successfully designed a memory management library with user-level page table and you are now an expert in virtual memory management. However, we have not thought about persistent storage. In this project, your task is to design an user-level file system with the FUSE filesystem library.

1. Description

File systems provide the 'file' abstraction. Standing between the user and the disk, they organize data and indexing information on the disk so that metadata like 'filename', 'size', and 'permissions' can be mapped to a series of disk blocks that correspond to a 'file'. File systems also handle the movement of data in to and out of the series of disk blocks that represent a 'file' in order to support the abstractions of 'reading' and 'writing' from/to a 'file'. You will be implementing a fairly simple file system called Tiny File System building on top of FUSE library.

2. FUSE Library

FUSE is a kernel module that redirects system calls to your file system from the OS to the code you write at the user level. While working with an actual disk through a device driver is a worthy endeavor, we would prefer you concentrate more on your file system's logic and organization instead of building one while also learning how to operate a disk directly. You will, however, have to emulate working with a disk by storing all data in a flat file that you will access as a block device. All user data as well as all indexing blocks, management information or metadata about files must be stored in that single flat file using whatever organization or format you wish. All your FUSE calls must only reference this 'virtual disk' and will provide the 'file' abstraction by fetching segments from the 'disk' in a logical manner. For example, if a call is made to read from a given file descriptor, your library would use the request to look up the index information to find out where the 'file' is located in your 'disk', determine which block would correspond to the offset indicated by the file handle, read in that disk block, and write the requested segment of the data block to the pointer given to you by the user. To give your file system implementation some context and scope, stub files will be provided.

3. Tiny File System Framework

You are given a code skeleton called Tiny File System structured as follow:

```
- block.c : Basic block operations, acts as a disk driver reading blocks from disk
- block.h : Block headers configures the BLOCK_SIZE
- tfs.c : User-facing file system structures and operations
- tfs.h : Contains Inode, Superblock, Direct structures. Also, provides
  functions for bitmap operations.
- Makefile
```

3.1 Block I/O layer

block.h/block.c specifies low-level block I/O operations. Since we're using a flat file to simulate a real block device (HDD or SSD), we use Linux read()/write() system call as our low-level block read and write operation. And you will be using the following two functions to implement Tiny File System functions on top of them.

`int bio_read(const int block_num, void *buf);`

Read a block at *block_num* from flat file (our 'disk') to *buf*

`int bio_write(const int block_num, const void *buf);`

Write a block at *block_num* from *buf* to flat file (our 'disk')

3.2 Bitmap

In *tfs.h*, you are given the following three bitmap functions:

`set_bitmap(bitmap_t b, int i)`

Set the ith bit of bitmap *b*.

`unset_bitmap(bitmap_t b, int i)`

Unset the ith bit of bitmap *b*.

`get_bitmap(bitmap_t b, int i)`

Get the value of ith bit of bitmap *b*.

In a traditional file system, both the inode area and the data block area needs a bitmap to indicate whether an inode or a data block is available or not (similar to virtual or physical page bitmap in project 3). When adding or removing a block, traverse the bitmap to find an available inode or a data block. Here are the two functions you need to implement:

`int get_avail_ino()`

Traverse the inode bitmap to find an available inode, set this inode number in the bitmap and return this inode number.

`int get_avail_blkno()`

Traverse the data block bitmap to find an available data block, set this data block in the bitmap and return this data block number.

3.3 Inode

Inode is the meta-data of a file or directory which stores important information such as inode number, file size, data block pointers; you could see the definition of *struct inode* in *tfs.h*. In Tiny File System, you only need to implement the following two inode operations:

`int readi(uint16_t ino, struct inode *inode)`

This function uses the inode number as input, reads the corresponding on-disk inode from inode area of the flat file (our 'disk') to an in-memory inode (*struct inode*).

`int writei(uint16_t ino, struct inode *inode)`

This function uses the inode number as input, writes the in-memory inode struct to disk inode in the inode area of the flat file (our 'disk').

3.4 Directory and Namei

Directory, in any kind of file system, is a very important component. Similar to Linux file systems, Tiny File System also organize files and directories in a tree-like structure. This means to look up a file or directory, your implementation of Tiny File System need to follow each pathname until the terminal point is found. For example, to look up "/foo/bar/a.txt", you will start at the root directory, find the mapping of **<inode number, file name>** in the data block of root directory to get the inode number of "foo", then find the mapping of **<inode number, file name>** in the data block of root directory to get the inode number of "bar", and then finally find the inode number of "a.txt", the terminal point of the path.

In *tfs.h*, you will find a directory entry called *struct dirent*. This struct describes the **<inode number, file name>** mapping of every file and sub-directory in the current directory. Thus, to create a file or directory in the current directory, one thing you need to do is to add a directory entry of the created file or directory. So is for delete a file or directory, you will have to remove the directory entry of the file or directory you want to delete in the current directory.

The following are the directory and namei functions you will have to implement:

```
int dir_find(uint16_t ino, const char *fname, size_t name_len, struct dirent *dirent)
```

This function takes the inode number of the current directory, the file or sub-directory name and the length you want to look up as inputs, and then reads all direct entries of the current directory to see if the desired file or sub-directory exists. If it exists, then put it into *struct dirent* dirent*.

```
int dir_add(struct inode dir_inode, uint16_t f_ino, const char *fname, size_t name_len)
```

In this function, you would add code to create a new directory. This function takes the current directory inode, the name along with its inode number as an input, then writes a new directory entry to the current directory. Look at the code skeleton in tfs.c for more hints.

```
int dir_remove(struct inode dir_inode, const char *fname, size_t name_len)
```

In this function, you would add code to remove a directory. To remove a directory, use the function's input: the current directory inode, the sub-directory name you want to delete, and then remove the directory entry in the current directory.

```
int get_node_by_path(const char *path, uint16_t ino, struct inode *inode)
```

This is the actual namei function which follows a pathname until a terminal point is found. To implement this function use the path, the inode number of the root of this path as input, then call *dir_find()* to look up each component in the path, and finally read the inode of the terminal point to *struct inode* inode. *For example, to look up '/home/Documents/apps/a.txt' this function resolves the path name, and finally reads the inode of "a.txt" into* struct inode *inode*

3.5 FUSE-based File System Handlers

As described in Section 2, the FUSE kernel module will intercept and redirect file system calls back to your implementation. In *tfs.c*, you will find a struct called *struct fuse_operations tfs_ope* which specifies file system handler functions for each file system operation (e.g., .mkdir = tfs_mkdir). After you mount the FUSE filesystem in the mount path, the kernel module starts redirecting basic filesystem operations (e.g., mkdir()) to TinyFS's filesystem handler (tfs_mkdir). For example, when you mount Tiny File System under "/tmp/mountdir", and you type the following command "mkdir /tmp/mountdir/testdir", the FUSE module

would redirect the call to *tfs_mkdir* instead of the default *mkdir* system call inside the OS. Here are the Tiny File System operations (handlers) you will implement in this project:

`static void *tfs_init(struct fuse_conn_info *conn)`

This function is the initialization function of Tiny File System. In this function, you will open a flat file (our "disk", remember the virtual memory setup) and read a superblock into memory. If the flat file does not exist (our "disk" is not formatted), it will need to call tfs_mkfs() to format our "disk" (partition the flat file into superblock region, inode region, bitmap region, and data block region). You must also allocate in-memory file system data structures.

`static void tfs_destroy(void *userdata)`

This function is called when your Tiny File System is unmounted. In this function, de-allocate in-memory file system data structures and close the flat file (our "disk").

`static int tfs_getattr(const char *path, struct stat *stbuf)`

This function is called when accessing a file or directory and provides the stats of your file, such as inode permission, size, number of references, and other inode information. It takes the path of a file or directory as an input. To implement this function, use the input path to find the inode, and for a valid path (inode), fill information inside *struct stat* stbuf*. We have shown a sample example.

`static int tfs_opendir(const char *path, struct fuse_file_info *fi)`

This function is called when accessing a directory (e.g., cd command). It takes the path of the directory as an input. To implement this function, fund and read the inode and if the path is valid, return 0 or return a negative value.

```
static int tfs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler,
                       off_t offset, struct fuse_file_info *fi)
```

This function is called when reading a directory (e.g., ls command). It takes the path of the file or directory as an input. To implement this function, read the inode and see if this path is valid, read all directory entries of the current directory into the input *buffer*. You might be confused about how to fill this *buffer*. Don't worry, in Section 7, we will give you some online resource as a reference.

`static int tfs_mkdir(const char *path, mode_t mode)`

This function is called when creating a directory (mkdir command). It takes the path and mode of the directory as an input. This function will first need to separate the *directory name* and *base name* of the path. (e.g., for the path "/foo/bar/tmp", the directory name is "/foo/bar", the base name is "tmp"). It should then read the inode of the *directory name* and traverse its directory entries to see if there's already a directory entry whose name is *base name*, and if true, return a negative value; otherwise, the *base name* must be added as a directory. The next step is to add a new directory entry to the current directory, allocate an inode also updating the bitmaps (more detailed hints could be found in the skeleton code).

`static int tfs_rmdir(const char *path)`

This function is called when removing a directory (rmdir command). It takes the path directory as an input. Similar to *mkdir*, this function will first need to separate the *directory name* and *base name* of the path. (e.g. for path "/foo/bar/tmp", the directory name is "/foo/bar", the base name is "tmp"). It then reads the inode of the *directory name*, and traverses the directory entries to see if there's a directory entry whose name is *base name*, if so, removes this directory from current directory, reclaims its inode, data blocks, and

updating bitmaps (more detailed hints can be found in the skeleton code). If the directory you want to delete does not exist, just return a negative value;

```
static int tfs_create(const char *path, mode_t mode, struct fuse_file_info *fi)
```

This function is called when creating a file (e.g., touch command). It takes the path and mode of a file as an input. This function should first separate the *directory name* and *base name* of the path. (e.g. for path "/foo/bar/a.txt", the directory name is "/foo/bar", the base name is "a.txt"). It should then read the inode of the *directory name*, and traverse its directory entries to see if there's already a directory entry whose name is *base name*, if so, then it should return a negative value. Otherwise, *base name* is valid file name to be added. The next step is to add a new directory entry to the current directory, allocate an inode, and update bitmap (more detailed steps could be found in skeleton code).

```
static int tfs_open(const char *path, struct fuse_file_info *fi)
```

This function is called when accessing a file. It takes the path of the file as an input. It should read the inode and if this path is valid, return 0, else return -1.

```
static int tfs_read(const char *path, char *buffer,
    size_t size, off_t offset, struct fuse_file_info *fi)
```

This function is the read call handler. It takes the path of the file, read size and offset as input. To implement this function, read the inode of this file from the path input, get the inode and the data blocks using the inode. Copy the desired data from data block from *offset* with *size* bytes to the memory area pointed by *buffer*.

```
static int tfs_write(const char *path, const char *buffer,
    size_t size, off_t offset, struct fuse_file_info *fi)
```

This function is the write call handler. It takes the path of the file, write size and offset as an input. To perform write, read the inode using the file path and using the inode, locate the data blocks, and then copy the data blocks to the memory area pointed by *buffer* from *offset* for *size* bytes.

```
static int tfs_unlink(const char *path)
```

This function is called when removing a file (rm command). It takes the path directory as an input. First, separate the *directory name* and *base name* of the path. (e.g. for path "/foo/bar/a.txt", the directory name is "/foo/bar", the base name is "a.txt"). Next, read the inode of the *directory name*, and traverse its directory entries to see if there's a directory entry whose name is *base name*, if so, then remove this directory from current directory, reclaim its inode, data blocks and updating bitmaps (more detailed hints could be found in the skeleton code). If the file you want to delete does not exist, just return a negative value;

3.6 Large file support

We know inode stores locations of data blocks of this file/directory. Different file systems use different ways to organize them. For example, FAT32 uses a file allocation table, Ext3 uses direct and indirect pointers, and Ext4 uses extents. In Tiny File System, we use a pointer-based mechanism (look at the definition of *struct inode* in *tfs.h*). However, the number of direct pointers in each inode is limited. To support large files, you must support single-level indirect pointers.

The indirect pointer works as follows: the data block pointed by an indirect pointer of an inode stores pointer to the "actual" data block. This mechanism is very similar to a one-level page table. For simply, you only need to implement 1-level indirect pointer. For directories (which holds all files and sub-directories as data), you *DO NOT* have to support indirect pointer.

4. Run Tiny File System

The code skeleton of Tiny File System is already well-structured. You will need to build and run Tiny File System on iLab machines. Unfortunately, not all iLab machines have FUSE library installed. We tested and list the following ilab machines that have already installed FUSE library:

```
cd.cs.rutgers.edu
cp.cs.rutgers.edu
ls.cs.rutgers.edu
kill.cs.rutgers.edu
```

Here are the steps to build and run Tiny File System.

1. Build Test Infrastructure (login to the above listed iLab machines)

   ```
   > mkdir /tmp/<your NetID>/
   > mkdir /tmp/<your NetID>/mountdir
   ```

2. Compile and Run Tiny File System

   ```
   > cd code
   > make
   > ./tfs -s /tmp/<your NetID>/mountdir
   ```

3. Check if Tiny File System is mounted successfully

   ```
   > findmnt (If mounted successfully, you could see the following information)

   /tmp/<your NetID>/mountdir   tfs    fuse.tfs    rw,nosuid,nodev,relatime,...
   ```

4. Exit and Umount Tiny File System

   ```
   > fusermount -u /tmp/<your NetID>/mountdir
   ```

To test the functionality of Tiny File System, you could just simply enter the mountpoint (In this case, it should be "/tmp/[your NetID]/mountdir"), and perform some commands (ls, touch, cat, mkdir, rmdir, rm ...). In addition, we also provide a simple benchmark (In the Benckmark folder in your skeletion code) to test your implementation of Tiny File System.

5. Debugging Tips for FUSE

1. Debugging in the FUSE library is not an easy step because we cannot simply use GDB to debug Tiny File System always. However, fuse provides **-d** options; when you run Tine File System with **-d** option, it will print all debug information and trace on the terminal window. Therefore, the best way is to add print statements to debug in combination with GDB for debugging your functions.
2. The fuse might return some errors (e.g. *Input/Output error* or *Transport endpoint is not connected*); this is because some FUSE file handlers are not fully implemented in the skeleton code. For example, if you have not implemented tfs_getattr(), a *cd* command into Tiny File System mountpoint will show errors.

6. Resources

`FUSE library API documentation`

```
    https://libfuse.github.io/doxygen/index.html

A FUSE file system tutorial:
    http://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/

A very useful tutorial:
    https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html
```

7. Submission and Report

You need to submit your Tiny File System code and a simple README file (specifies the all the member's name and NetID in your project group) in a compressed tar file on Sakai. You could use the following command to archive your Tiny File System folder named code. Your report will include details on the total number of blocks used when running the sample benchmark, the time to run the benchmark, briefly on how you implemented the code, and finally, any additional steps that we should follow to compile your code.

```
    > tar -zcvf <your_net_id>.tar.gz code
```

8. Ideas and thought on Tiny File System

In this project, you will implement a workable file system all on your own. Please think about the following questions:

```
1. How many times bio_read() is called when reading a file "/foo/bar/a.txt" from
   our 'disk' (flat file)?

2. How to reduce the number of bio_read() calls?

3. Besides storing meta-data structures in memory, what else could you do to
   improve performance?

4. In each API function of the skeleton code, we have provided several steps.
   Think about what would happen if a crash occurs between any of these steps?
   How would you improve the crash-consistency of Tiny File System?
```

You don't have to submit anything about these questions. Just think about them when you finish your project.

```
- Your grade will be reduced if your submission does not follow the above instruction.
- Borrow ideas from online tutorials but DO NOT copy code and solutions, we'll find out.
```

Created by Yujie REN and modified by CS416 staff.