CS 416/518 Project 4: RU File System using FUSE

Due: April 29th 2024, at 8 am (No Extensions, Start Early!)

Points: 100

In this project, your task is to design a user-level file system with the FUSE file system library. In addition, we have provided a simple test case (*simple_test.c*) and a detailed test case (*test_case.c*) to help you debug and guide you as you develop your code. We have also provided some *crucial* debugging tips for FUSE. Finally, FUSE-based file systems are used in several real-world applications including Android-based phones. You are building a real user-level file system; so write clean code, modularize your code, and add code comments where possible.

Please take some time to read the instructions carefully before getting started with implementation. You should approach your implementation like any seasoned software developer would; list down things you understand, list down things you do not understand, and take time to come up with some high-level thoughts on how you will go about designing the solution.

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 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 RU File System (RUFS), building on top of the 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 prefer that you concentrate more on your file system's logic and organization in a user-level filesystem, rather than dealing with the complexities of learning how to operate a disk correctly. You will, however, emulate a working disk by storing all data in a flat file that you will access like 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.

All your FUSE calls will 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.

The iLab machines are already running the FUSE driver. On the class VM, you can install the FUSE driver by running the commands

```
sudo apt-get update
sudo apt-get install libfuse-dev
```

So, when you run RUFS, the current code registers important file system functionalities that must be handled with the FUSE driver. How is that done? Look in the *rufs_ope* structure in rufs.c, which registers with the OS FUSE driver the functions you will be handling. The *.init*, *.destroy*, and other structure variables are generic stubs that any new file system must implement.

More details on FUSE can be found in the Resources section (Section 6).

```
static struct fuse_operations rufs_ope = {
  .init   = rufs_init,
  .destroy  = rufs_destroy,
```

```
  .getattr  = rufs_getattr,
  .readdir  = rufs_readdir,
  ...
}
```

3. RU File System Framework

The skeleton code of RUFS is structured as follows:

- *code/block.c* : Basic block operations, acts as a disk driver reading blocks from disk. Also configures disk size via *DISK_SIZE*
- *code/block.h* : Block layer headers. Also configures the block size via *BLOCK_SIZE*
- *code/rufs.c* : User-facing file system operations
- *code/rufs.h* : Contains inode, superblock, and dirent structures. Also, provides functions for bitmap operations.
- *code/benchmark/simple_test.c* : Simple test case for initial development.
- *code/benchmark/test_cases.c* : More comprehensive test cases.
- *Makefile* : in both ./ and ./benchmark/ . Similar to the VM project, first compile the RUFS code and then the benchmark code.

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. You will be using the following two functions to implement RUFS functions on top of them. We have provided the implementation of *bio_write* and *bio_read* functions.

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

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

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

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

Note there are also three other functions to help with your flat file 'disk':

`void dev_init(const char* diskfile_path)`

Creates a flat file at the given path. The file will be of size *DISK_SIZE* bytes and will serve as your 'disk'. This should be called if the file hasn't been created yet.

`int dev_open(const char* diskfile_path)`

Simply opens the flat file that will serve as the 'disk'. This should be called upon mounting the file system, so the 'disk' could be read from and written to using *bio_read()* and *bio_write()*

`void dev_close()`

Simply closes the *disk* file. This should be called when the file system is being unmounted.

Note: There is a global variable called *diskfile_path* in rufs.c. This variable is set in the main() function within rufs.c, setting the path to `<current working directory>/DISKFILE`. You should use this path for your disk file and pass this global variable into dev_init() and dev_open(). It is done this way to avoid hardcoding a path for the disk file, making rufs more portable. Do not set *diskfile_path* yourself.

3.2 Bitmap

In *rufs.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

An inode is the meta-data of a file or directory that stores important information such as inode number, file size, data block pointers; you could see the definition of *struct inode* in *rufs.h*. In RUFS, 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

Directories, in any file system, is an essential component. Similar to Linux file systems, RUFS also organizes files and directories in a tree-like structure. To lookup a file or directory, your implementation of RUFS must follow each part of the path name until the terminal point is found. For example, to lookup "/foo/bar/a.txt", you will start at the root directory ("/"), look through the directory entries stored in the data blocks of the root directory to get the inode number of "foo", then look through the directory entries stored in the data blocks of the directory "foo" to get the inode number of "bar", and then finally look through the directory entries stored in the data blocks of the directory "bar" to get the inode number of "a.txt", the terminal point of the path.

In *rufs.h*, you will find a directory entry structure 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/sub-directory in the current directory, you will need to add a directory entry for created file/sub-directory in the current directory's data blocks. Likewise, to delete a file/sub-directory from the current directory, you will have to remove/invalidate the directory entry of the file/sub-directory in the data blocks of 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 lookup 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 add a new directory entry. This function takes as input the current directory's inode stucture, the inode number to put in the directory entry, as well as the name to put into the directory entry. The function then writes a new directory entry with the given inode number and name in the current directory's data blocks. Look at the code skeleton in rufs.c for more hints.

```
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 lookup each component in the path, and finally read the inode of the terminal point to "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 *rufs.c*, you will find a struct called *struct fuse_operations rufs_ope* which specifies file system handler functions for each file system operation (e.g., .mkdir = rufs_mkdir). After you mount the FUSE filesystem in the mount path, the kernel module starts redirecting basic filesystem operations (e.g., mkdir()) to RUFS handler (rufs_mkdir). For example, when you mount RUFS under "/tmp/mountdir", and you type the following command "mkdir /tmp/mountdir/testdir", the FUSE module would redirect the call to *rufs_mkdir* instead of the default *mkdir* system call inside the OS. Here are the RU File System operations (handlers) you will implement in this project:

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

This function is the initialization function of RUFS. 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 rufs_mkfs() to format our "disk" (partition the flat file into superblock region, inode region, bitmap region, and data block region). You must also allocate any in-memory file system data structures that you may need.

```
static void rufs_destroy(void *userdata)
```

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

```
static int rufs_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 in the skeleton code. On success, the return value must be 0; otherwise, return the right error code.

Note 1: If you do not implement rufs_getattr(), a *cd* command into RUFS mountpoint will show errors. Other parameters to fill include: st_uid, st_gid, st_nlink, st_size, st_mtime, and st_mode. The template code already shows how to do it for some parameters. You would have to do it for all files and all directories including the root. For setting st_uid and st_gid, you could use getuid() and getgid(). For other parameters, think!

Note 2: rufs_getattr() must also be called before creating a file or directory to check whether the file or the directory you want to create already exists. So, think about what return value you would expect from rufs_getattr().

```
static int rufs_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, find and read the inode, and if the path is valid, return 0 or return a negative value.

```
static int rufs_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 rufs_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, and also update the bitmaps (more detailed hints could be found in the skeleton code).

```
static int rufs_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 a valid file name to be added. The next step is to add a new directory entry ("a.txt") using *dir_add()*
to the current directory, allocate an inode, and update the bitmaps (more detailed steps could be found in skeleton code).

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

This function is the read operation's 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 *size* bytes from the inodes data blocks starting at *offset* to the memory area pointed to by *buffer*.

```
static int rufs_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 a write operation, read the inode using the file path and using the inode, locate the data blocks and then copy *size* bytes from the memory area pointed by *buffer* to the file's data blocks starting at *offset*.

Note 3: All operations in the RUFS (and any file system) are done in the granularity of "BLOCK_SIZE." So, if you want to update only a few bytes of a block (say 5 bytes in inode blocks or bitmaps or superblock), you must read the entire disk block to a memory buffer, apply your change, and write the in-memory buffered block to the disk block.

Note 4: You must handle unaligned write/read (i.e., writing to or reading from an offset that is not aligned with the BLOCK_SIZE).

4. (CS416: Not Required) (CS518: 20 points)

`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 entry from a directory. This function takes as input the current directory's inode structure and the name of the directory entry you want to remove. To remove a directory entry, you would look in the data blocks of the given directory for the directory entry with the corresponding name, then mark the directory entry as invalid.

`static int rufs_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 the current directory, reclaims its inode, data blocks, and updates the 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 rufs_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/invalidate this directory entry in the data blocks of the current directory, reclaim the inode, data blocks of the inode, and update the appropriate 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;

Remember, you are not required to support indirect pointers, but you should try and implement this if you finish the basic requirements of the project and time allows.

---

5. Run RUFS

The code skeleton of RUFS is already well-structured. You could build and run RUFS on iLab machines or the class-provided VMs. If you're using ilabs though, Unfortunately, not all iLab machines have the FUSE library installed. If you end up using ilab, the following ilab machines have been tested and known to have the FUSE library already installed:

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

Here are the steps to build and run RUFS.

1. Login to one the iLab machines listed above
2. Make a directory under */tmp/* to mount your file system to

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

3. Compile RUFS by naviagating to your code and running *make*

```
$ cd code
$ make
```

4. Mount RUFS to the directory you just created

```
$ ./rufs -s /tmp/<your NetID>/mountdir
```

5. Check if RUFS is mounted successfully using *findmnt*

```
$ findmnt
```

If mounted successfully, you could see the following information in the output of *findmnt*:

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

If you want to stop the RU File System, you can exit and umount RUFS by running the following:

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

6. Testing RU File System Functionality

To test the functionality of RUFS, 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 benchmarks (In the Benckmark folder in your skeletion code) to test your implementation of RU File System. To run the benchmarks:

1. Make sure RUFS is mounted and running.
2. Ensure to benchmarks know where to test the file system by configuring *TESTDIR* in each benchmark to point your file system's mount point (ex. /tmp/[your NetID]/mountdir)
3. Compile the benchmarks using the provided Makefile
4. Run the benchmarks

If you skip the optional directory and file remove operations, running the benchmarks a second time may fail, since the creation operations fail when the files/directories already exist. If you encounter this issue, it could help to unmount, delete the `DISKFILE`, and try running the benchmark again.

7. Debugging Tips for FUSE

1. Debugging in the FUSE library is not an easy step because we cannot simply use GDB to debug RUFS always. However, FUSE provides **-d** options; when you run RUFS with the **-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 "-d" parameter is supposed to run RUFS continuously, which gives a notion that your file system is hanging. So, when you debug using "-d," open a SEPARATE terminal window and run your commands from there. The Fuse system with "-d" also has another caveat that if you CTRL-C out of the -d "hang," it also unmounts the file system, which is why you will not see your file system mounted (when using "findmnt").
3. If you want to run RUFS in the foreground, use **-f** option if (2) is insufficient. For getting more help on commands use,

```
$ ./rufs --help
```

Note: 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 rufs_getattr(), a *cd* command into the RUFS mount point will show errors.

8. 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/

Two very useful tutorials:

- https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/
- https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html

9. Suggested steps for designing a solution

Given the nature of the project it is quite hard to have a suggested steps. Its hard because in order to start testing out things, using commands like "cd,mkdir,ls,touch" etc.. all the functions have to be implemented for the most part.

However we will say the following steps:

1. rufs_mkfs, rufs_init, and rufs_destroy should probably be the first place to start as you need a "disk" to even start
2. You should probably implement all the helper functions as you will make good use of them when implementing the rest of the rufs functions.
3. The bitmap operations are the easiest to implement
4. The inode helper operations (readi and writei) are the second easiest
5. The directory operations are the hardest one
6. Once you implement some of the helper functions, you can start using them within the rest of the rufs functions you have to implement.

Solving the project requires understanding file system concepts and understanding what functions are used where. If you do not have a firm understanding of how the file system is laid out or how directories work, you will have trouble trying to implement because you may not know at the start what the purpose of this function is or when it will be used.

One thing we would suggest is to take a paper and pen and go through the operations that need to be done to do a certain operation for example, finding a file within a particular directory or creating a simple file called "a.txt" within the root directory.

10. Submission and Report

Submit the following files as one Zip file in Canvas:

1. rufs.c
2. rufs.h
3. Makefile
4. Any other supporting source files you created
5. A report in .pdf format

The report should contain the following things:

- Partners names and netids
- Details on the total number of blocks used when running the sample benchmark, the time to run the benchmark, and briefly explain how you implemented the code
- Any additional steps that we should follow to compile your code

- Any difficulties and issues you faced when completing the project
- If you had issues you were not able to resolve, describe what you think went wrong and how you think you could have solved them
- Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

11. Ideas and thoughts on RUFS

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

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

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

Remember:

- 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 will find out.