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

Due: 12/10/2025; 11:59pm

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 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.

In the interest of time, for this project you are only expected to support a few basic operations, such as creating, reading, and writing small files, as well as handling directories. Other important functions, such as file or directory deletion and several additional file system features, can be skipped. Please read on for details. You do not have to support multithreaded applications.

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 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 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 lookup 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. 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 8).

```
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:

File		Description
`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 and 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/test_cases.c`	-	More comprehensive test cases.
`Makefile`	-	In both code/ and code/benchmark/. Similar to VM project, first compile RUFS code and then the benchmark code.

3.1 Block I/O layer

Files 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'll use Linux's read() and write() system calls as our low-level block read and write operations. 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 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. You should not set *diskfile_path* yourselves.

3.2 Bitmap

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

- `set_bitmap(bitmap_t b, int i)` - Set the *i*th bit of bitmap *b*.
- `unset_bitmap(bitmap_t b, int i)` - Unset the *i*th bit of bitmap *b*.
- `get_bitmap(bitmap_t b, int i)` - Get the value of *i*th bit of bitmap *b*.

In a traditional file system, the inode area and the data block area need their own 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 the bit, and return the inode number.

- `int get_avail_blkno()`

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

3.3 Inode

An inode is the meta-data of a file or directory that stores important nformation 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.

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. Project Parts That Can Be Skipped

The following parts can be skipped in the interest of time.
All of them are already present as stubs in `rufs.c` with:

```
// For this project, you don't need to fill this function
// But DO NOT DELETE IT!
```

You do not need to modify these stubs for grading.

4.1 Optional FUSE Handlers that you don't have to implement (and will not be graded)

- `static int rufs_rmdir(const char *path)`

  This handler would remove a directory for the `rmdir` command. It would:

  1. Split `path` into parent directory and base name (e.g., for `/foo/bar/tmp`, parent is `/foo/bar`, base is `tmp`).
  2. Look up the target directory inode in the parent.
  3. Ensure the directory is empty.
  4. Remove the directory entry from the parent, reclaim the inode and data blocks, and update bitmaps.

  You are **not required** to implement this handler.

- `static int rufs_unlink(const char *path)`

  This handler would remove a file for the `rm` command. It would:

  1. Split `path` into parent directory and base name (e.g., for `/foo/bar/a.txt`, parent is `/foo/bar`, base is `a.txt`).
  2. Look up the file inode via the parent directory.
  3. Invalidate/remove the corresponding directory entry in the parent's data blocks.
  4. Reclaim the inode and its data blocks and update the appropriate bitmaps.
  5. Return a negative error code if the file does not exist.

  You are **not required** to implement this handler.

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

  Called when a directory handle is released. The stub that simply returns `0` is sufficient for this project. You do **not** need to add logic here.

- `static int rufs_truncate(const char *path, off_t size)`

  Would change the logical size of a file (e.g., truncation). The stub that simply returns `0` is sufficient. You do **not** need to implement truncate logic.

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

  Called when a file handle is released. The stub that returns `0` is sufficient. You do **not** need to implement additional logic here.

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

  Would flush cached file data to disk. The stub that returns `0` is sufficient. You do **not** need to change it.

- `static int rufs_utimens(const char *path, const struct timespec tv[2])`

Would update file timestamps (access/modify times). The stub that returns `0` is sufficient for this project.

You are also **not required** to support indirect pointers. So, this means, we are only supporting small files.

5. Running RUFS

The code skeleton of RUFS is already well-structured. You will need to build and run RUFS on iLab machines. Unfortunately, not all iLab machines have the FUSE library installed. 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,...
   ```
6. 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 . . .).

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

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 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 to can start to use 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

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.

12. Frequently Asked Questions (FAQs)

Debugging Questions

- **What is the `-s` flag in the command `$ ./rufs -s /tmp//mountdir`?**

  The `-s` flag indicates single-threaded instead of multi-threaded. This makes debugging vastly easier, since gdb doesn't handle multiple threads all that well. It also protects you from all sorts of race conditions. Unless you're trying to write a production filesystem and you're a parallelism expert, I recommend that you always use this switch. For more info see: https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html

Open/Create/Initialize Related Questions

- **Do we need to handle flags for rufs_open()? Like prohibit writing of a file is opened for RDONLY?**

  No. In this project for rufs_open, you are only required to just check if the file exists.

- **Should we return -1 if a file does not exist but the user is trying to open the file?**

Instead of returning -1 upon a file not being found, try including errno.h and return the ENOENT error code which is the error code that corresponds to "no such file or directory found" like the following code snipper. You can look at is more here: http://man7.org/linux/man-pages/man3/errno.3.html

```
#include <errno.h>
....

if(inode not found){
    return ENOENT;
}

// or

if(inode not found){
  return -ENOENT;
}
```

- **How to update superblock, inode bitmaps and data bitmaps?**

  First regarding the superblock, when doing rufs_mkfs(), you will allocate space for a superblock and then we have to write a block to the disk for our superblock. Since the superblock will be the first block in the disk, anytime you update the superblock, you will need to use the bio_write() function to write our superblock to block 0.

  Same goes for updating inode bitmaps and data bitmaps, and inode blocks and data blocks. You would have to use bio_write().

- **The stat structure has a lot of possible fields to fill, and I was wondering which ones are necessary?**

  ```
  int(* fuse_operations::getattr)(const char *, struct stat *, struct fuse_file_info *fi)
  ```

  Get file attributes. Similar to stat(). The 'st_dev' and 'st_blksize' fields are ignored. The 'st_ino' field is ignored except if the 'use_ino' mount option is given. In that case it is passed to userspace, but libfuse and the kernel will still assign a different inode for internal use (called the "nodeid").

- **What is the units of Size? Is it the number of valid dirents in a director? The number of blocks used in direct_ptr?**

  So the size member refers to the size of the actual file. You can use blocks or bytes, it's up to you. Just remember if you happen to use one or the other, you may have to convert it to fill in the following fields in struct stat in rufs_getattr():

  ```
  off_t     st_size;        /* Total size, in bytes */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  ```

- **What is the inode stat thing, and what is its purpose?**

  We use struct stat to record the time related to an inode (the "vstat" field in struct inode in rufs.h). When a file is created or modified, we need to update this field in its inode. For more information about struct stat, please see "https://linux.die.net/man/2/stat". In this project, we ONLY use st_mtime and st_atime in struct stat. You could use function time() to set st_mtime and st_atime. For more information and example about time(), please see "https://linux.die.net/man/2/time".

- **What exactly is a link (i.e., link count) and what benefit does knowing how many there are have?**

  This "link count" value is the number of different directory entries that all point to the inode of a file or subdirectory. In the case of a regular file, the link count is the number of hard links to that file (which is generally one). However, for a directory, even for the empty directory's inode, you will have a link count 2. In Linux or OSx you can find the number of links for a file or directory using the following command.

```
$ ls -a file1
$ mkdir emptydir1    //create empty directory.
$ ls -a emptydir1
```

- **Why 2 links for directories?**

  Every directory also contains the "." link that points back to itself. So the minimum value of 2 links per directory. Every subdirectory has a ". . . " link that points back to its parent, incrementing the link count on the parent directory by one for each subdirectory created. So, when you created "emptydir1", the link count of the parent would have incremented by 1.

- **Where else to update the link values?**

  See the stat structure carefully, which has a st_nlink member variable. This must be also updated. Return type of rufs_init The return value of this function is available to all file operations in the private_data field of fuse_context. It is also passed as a parameter to the destroy() method (See the link mentioned in Resources section 7 of the description). If you don't have anything to pass as a privata_data (or store some state to be used by all FS operations), you could just return a NULL (in rufs_init).

- **What is a direct_ptr in the inode struct, and the member, vstat?**

  The direct_ptr(s) are a list of pointers to the data blocks of this particular file represented by this inode. By pointers, we mean they store the block numbers of the data blocks. vstat is simply a struct holding file info that you will have to keep for getattr. Look at the man page for struct stat here (http://man7.org/linux/man-pages/man2/stat.2.html)

- **If a direct_ptr has 16 elements (entries), should we assume every file/directory is going to take up 16 data blocks?**

  No, 16 represents the maximum blocks that could be supported with direct pointers in this project. There could be files that take fewer than 16 data blocks.

Root Directory Questions

- **Does "/" represent the root directory? Do we need a dirent struct for root?** Yes, the initial "/" is the root, you do not necessarily need a dirent struct for it but you may need an inode to represent it.

- **Will all paths start with '/'? Is there a chance to have something like "~/someDir/someDir2" or "../" or "."?**

  While in real file systems this is absolutely possible, for this project you don't have to worry about the relative path or path w.r.t to current working directory. When we test, we will use full path strings.

  If you are curious about how path resolution works in a real file system, here are some details, http://man7.org/linux/man-pages/man7/path_resolution.7.html

- **What should the mode be for the root directory node?**

  You can probably set it to something like: `S_IFDIR | 0755`

General Directory Questions

- **What is the difference between data blocks and directory blocks?**

  The data blocks of file inodes simply contain users data (i.e., contents of a file) and no dirents (directory entries are stored) The data blocks of directory inodes contain dirents structure.

- **How to represent and store directories?**

  A directory is also an inode with direct pointers pointing to data blocks. In a directory's data blocks, you store instances of struct dirent. The maximum number of dirents stored in a directory is limited by the number of data blocks that can be used by directory inode, block size, and the size of each dirent. Because we are only using direct data block pointers (16 blocks in this project), you can easily calculate the number of dirents you could store in 16 data blocks.

- **Iterating through the block: How do we know how many data blocks the current directory is using?**

  In order to know how many data blocks a current directory is using, you can just keep track of the size of the directory like you would a regular file as you add dirents and allocate more data blocks to hold those dirents. As for which direct ptrs are valid or not, you can set the direct_ptr to some default value when not in use, like 0, as we know block zero we reserved for the superblock.

- **Since by default there is no structure in the data block, should we make the desired data block "struct aligned" (iterate through the data block one dirent at a time, initialize all entries and set each valid in a dirent structure to 1 {is this how valid should be used?} until the block is full) once the first directory entry of this block is being created?**

  Initially, when a new data block is allocated for a directory, it should be formatted so that all entries are set to invalid, that way the entires can be skipped when you have to iterate over the directory entries, like in dir_find. Then, when the first directory entry of a block is added, then only that one entry should be marked as valid. As for how many dirents you will have to have per block is something you can figure out by finding the size of a struct dirent, and seeing how many dirents you can fit inside a single 4k block.

- **Is there a reason for the size of the dirent to be a number as ugly as 214? May we increase the size of the name buffer to make the total size of the struct a round number like 256? This would make the math easier when reading/manipulating directories.**

  It is okay to increase/pad the struct dirent to 256 bytes for this project. You will not lose points. Beyond this project, just because doing the math is complex, one cannot introduce internal fragmentation. Imagine a disk with 10-million directories, so that's around 400MB of disk space. While real FSes do try to align the structures in powers of 2, unfortunately, it is not always possible.

- **Do we need to support directories that contain a lot of files so that one data block doesn't hold all it's entries?**

  If you are only supporting direct pointers and not doing the extra credit, a directory shoyuld be able to support as many directory entries as can fit in 16 data blocks

File Read and Write Questions

- **What is the purpose of offsets?**

  The offset it simply the starting offset within the whole file. For example, let's say I have a 16kb file called "foo" that I want to read or write to. Since each block is 4k, this means foo's inode points to 4 data blocks. Now lets say you want to write a 5 byte string "hello" to file foo starting at offset 6144. `const char *str = "Hello";   rufs_read("foo", str, 5, 6144, asdasdas)` So starting at offset 6144 within the file, I should start to copy the string hello. Since offset 6144 would lie in the middle of the second block (6144 - 4k in the first block = offset 2048 in the second block), that is where I would have to start writing. So I would read in the second block, copy the data, then write the second block back to disk.

Locking Questions

- **Do we need to implement some kind of locking to FS functions?**

Not needed for this project.

Destroy Operations Questions

- **In rufs_destroy the hint says "De-allocate in memory data structures". What exactly should we do here?**

  Release all the malloc'd memory (if you did allocate).

Submission Questions

- **Is okay for us to modify block.c/block.h?**

  If you are adding helper functions, as long as you state it clearly in the report and upload them alongside with the required files, that's fine.

- **Is there a specific output that we have to be looking for when we run our code in the simple_test.c file?**

  There are seven things that simple test.c (take a look at the simple test code carefully to see what it's doing) so you should so hopefully is correctly implemented, you should see success for each case like:

  ```
  Test 1: Success

  Test 2: Success

  etc...
  ```