CS 454/654 **Project 1 and 2**

See LEARN for deadlines and TA office hours.

Read this document *carefully* and note the important sections. You will need to refer back to this document *frequently* when implementing P1 and P2.

Very Important: You will need to dedicate sufficient time for P1 and P2—to **design, implement, and most importantly debug** your code! A last weekend dash is unlikely to allow sufficient time to finish the complete Project, especially P2.

Contents

1 Introduction

In these **projects**, you will implement *WatDFS*, a simplified distributed file system. WatDFS will act as a transparent layer on top of the local file system to support creating, opening, reading, writing, and closing files on remote machines. The **Project** comes in 2 parts. In **P1**, you will first implement WatDFS using a remote access model (Section 6). In **P2**, you will extend **P1** to support a download/upload model with client-side caching (Section 7).

To implement the WatDFS file system, you will integrate with FUSE,[1] a popular library that enables the development of custom file systems. We will provide a library that takes care of setting up the FUSE functionality for you. We will also provide an RPC library that you will use to make remote procedure calls between the WatDFS client and server.

This document presents an overview of FUSE (Section 2) and describes how you should integrate your solution with FUSE (Section 3). It also describes the API of the provided RPC library (Section 4), tips for doing the **projects** (Section 6 and 7), and pointers to more resources (Section 5). Section 8 contains answers to frequently asked questions and solutions to common problems you may encounter while implementing WatDFS.

Ensure that you have read all the relevant parts of this document before beginning the projects as it contains answers to many frequently asked questions and tips to avoid common pitfalls during implementation.

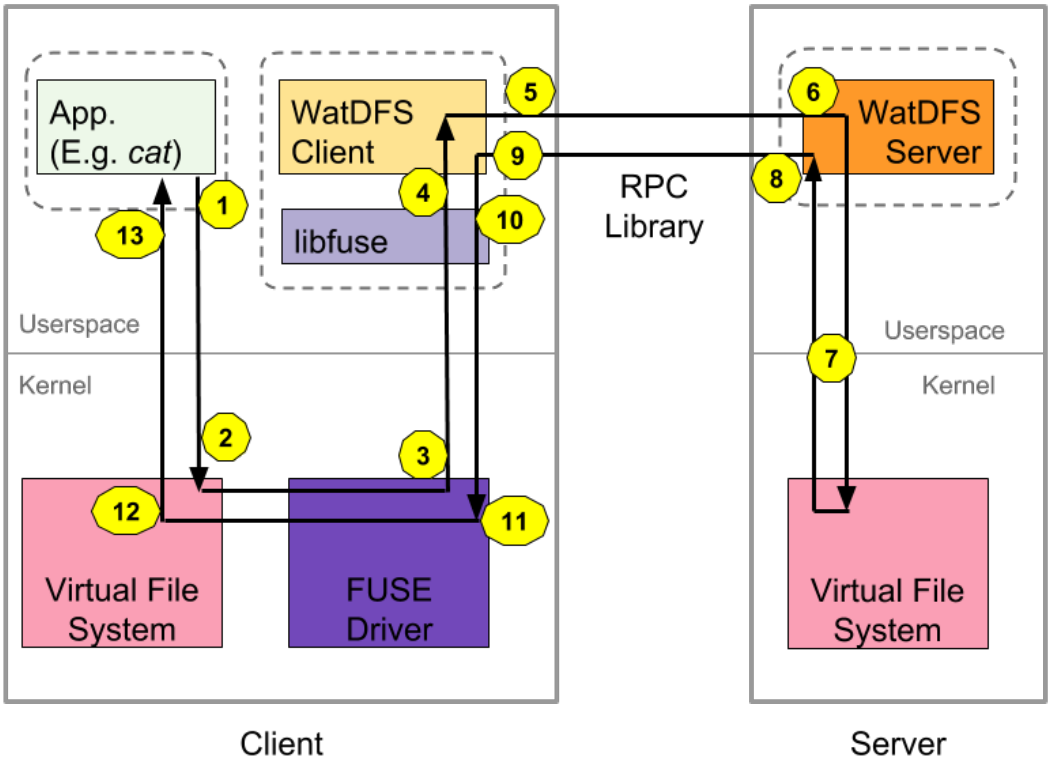
[1] https://github.com/libfuse/libfuse

2 WatDFS Overview

Distributed file systems like Coda and NFS allow a user to interact with files that may be located on a remote server. To do so, the distributed file system presents an interface that acts as if remote files are located on the local file system and are therefore accessible through standard system calls like `open`, `close`, `read`, and `write`.

FUSE presents the interface of a standard file system, and therefore maintains access transparency. Additionally, FUSE requires no kernel code modification. FUSE consists of two parts: the FUSE kernel driver and the library *libfuse*. Libfuse integrates with the Linux *virtual file system* and passes calls on to a user space library such as WatDFS, the distributed file system that you will implement.

Below is a diagram that traces the steps (indicated by numbered arrows) that a file system operation (such as reading a file) invoked by an application (e.g., the `cat` command) will take to reach WatDFS, as well as the actions that WatDFS should take. The dashed lines indicate separate processes, the thick gray lines separate machine boundaries, and the thin gray lines separate user space from the kernel. You will be responsible for implementing the WatDFS client and WatDFS server components.



1. An application (e.g., `cat` or `os.open()` in Python) performs a file system operation by making a system call (such as `open()`, `close()`, `read()`, or `write()`). This system call is passed to the Linux kernel.
2. The kernel redirects the system call to the virtual file system (VFS).
3. In general, the VFS transforms the system call into an appropriate call against the file

system indicated by the file path. In WatDFS, the VFS will detect that the file system call is to be served by WatDFS, a FUSE-based file system. Consequently, the call is passed to the FUSE kernel driver.

4. The FUSE kernel driver delivers the call to *libfuse*, which will pass the calls on to the WatDFS client library that you'll implement.
5. The WatDFS client library makes an RPC call to the WatDFS RPC server with details about the system call being performed.
6. The WatDFS RPC server receives the RPC call and calls the appropriate functions in the WatDFS server implementation.
7. The WatDFS server performs the indicated system call against the local file system and receives a response from the file system.
8. The WatDFS server responds to the RPC call, sending back data and return codes from the system call.
9. The RPC response is passed back to the WatDFS client library and the client library generates a response to the FUSE driver's request.
10. The WatDFS client library forwards the FUSE response to the FUSE driver.
11. The FUSE driver passes the response back to the VFS.
12. The VFS translates the file system response back into a system call response for the kernel.
13. The application receives the result of the system call.

To complete P1 and P2, you must implement steps 5–10 for various file system operations (described in Section 3) with the assistance of the provided libraries. To test your system locally, you will need to use command line applications (e.g., `cat`, `stat`) or write small C++ or Python code snippets (e.g., `os.read()`) that can perform end-to-end tests indicated by steps 1–13. When you submit your code, we will test the WatDFS client and server functionality that you've written both individually and with complete end-to-end tests. This testing of your code will be done through automated tests on Marmoset. **Consequently, it is important that you implement the projects exactly as this document specifies.**

WatDFS execution architecture: Only one application (e.g. `cat`) will interact with your WatDFS client at a time.[2] However, there may be multiple WatDFS client instances that are connected to the same WatDFS server. In other words, the WatDFS client has to process only one request at a time, while the WatDFS server needs to account for multiple concurrent requests.

Integrating with FUSE requires mapping the FUSE kernel driver's calls to functions in the client WatDFS library. We have implemented this mapping, while filtering out file system calls that are not relevant to the projects (e.g. directory operations). Your task is to implement the WatDFS library functions needed to support key FUSE operations, which we describe in the next section.

[2]This restriction simplifies development for these projects; FUSE supports multiple client applications interacting with the same user space *libfuse* library.

3 FUSE Operations to Support

To support opening, reading, writing and closing files, you will be required to support several FUSE functions that are grouped below into categories of related operations. You will need to consult the Linux man pages to find the corresponding system call that must be made to support each *libfuse* function. Some pointers on helpful system calls are provided in Section 5.2. Below, we also provide links to the related *libfuse* calls that provide more details on the specifications of the functions.

3.1 General Notes

In general, the calls made by FUSE should return `0` on success, or `-errno` if an error occurs (the exceptions are `read` and `write`). **This behaviour differs from system calls, which typically return `0` on success and `-1` on error.** If an error occurs, system calls set `errno` to a positive number to indicate what went wrong. You can find the complete list of `errno` in errno-base.h and errno.h.

libfuse always provides path names of the file it wishes to operate on, but using names rather than file descriptors results in more expensive lookups and metadata tracking. Therefore, some libfuse functions take a `fuse_file_info` structure. This structure has an integer member variable `fh`, which you should set in the `open` function call and use afterwards to identify the file (i.e., act as a file descriptor).[3]

```
struct fuse_file_info {
    int flags; // open flags, available in open and release.
    /* other fields */
    uint64_t fh; // file handle, may be filled in by open
};
```

The path argument is **relative to the directory** where FUSE is mounted.

The FUSE function calls you will implement accept an additional `userdata` argument that provides contextual and state information, which we describe shortly. If you are uncertain about other arguments and their types used in the (e.g., `mode_t`), additional information can be found by searching the *libfuse* headers[4] or Linux headers.

3.2 Initialization and Destruction

These two functions are used to initialize the *libfuse* integration and cleanup on shutdown respectively.

3.2.1 `watdfs_cli_init`

```
void *watdfs_cli_init(struct fuse_conn_info *conn,
                      const char *path_to_cache, time_t cache_interval
```

[3] Although you could implement these projects using only the file names that are provided in the function arguments, you **must** use the `fuse_file_info` for both performance and compatibility with the automated tests.

[4] libfuse/fuse.h

```
                        int *ret_code);
```

This function is called during the initialization of the file system. This function can be used to perform a one-time set up and initialization.[5]

The `void *` pointer that is returned from this function will be passed to every other function call as the `userdata` argument. `userdata` is a global pointer variable that can be used to store state. You should allocate and initialize any global data structures that you need for the WatDFS client inside `watdfs_cli_init`.[6]

Because the return value from the function is `userdata`, the `ret_code` argument is used to indicate success or failure instead. You must set `ret_code` to `0` on success or an appropriate non-zero value on failure. *libfuse* will exit if `ret_code` is not set to `0`.

We will describe the `path_to_cache` and `cache_interval` arguments in P2. You can ignore them for P1.

3.2.2 watdfs_cli_destroy

```
void *watdfs_cli_destroy(void *userdata);
```

This function is called when the file system is being destroyed, and should free any allocated structures during initialization.

3.3 File Attributes

Several file system operations require metadata information about a file (e.g., file size, number of blocks) as defined by the `stat` structure. The functions below fill in the `stat` structure by getting attributes for the file given in `path`.

Important: *libfuse* checks the output `stat` structure (`statbuf`) to determine whether a file exists. If the file does not exist, do not fill in the `statbuf` structure.

3.3.1 watdfs_cli_getattr

```
int watdfs_cli_getattr(void *userdata, const char *path,
                       struct stat *statbuf);
```

[*libfuse* reference: fuse.h#L300-L311]

3.4 Opening and Closing

3.4.1 watdfs_cli_mknod

```
int watdfs_cli_mknod(void *userdata, const char *path,
                     mode_t mode, dev_t dev);
```

[*libfuse* reference: fuse.h#L323-L329]

This function is called to create a file if it does not exist.

[5] `watdfs_cli_init` is not responsible for starting the file system. Our provided client code will initialize and mount the FUSE file system, and then `watdfs_cli_init` will be called.

[6] `userdata` can be a useful global structure in P2, but you will not need it for P1.

Important: If an application calls `open` with the `O_CREAT` flag and the file does not exist, `watdfs_cli_mknod` is called by FUSE before the actual `watdfs_cli_open` call.

3.4.2 watdfs_cli_open

```
int watdfs_cli_open(void *userdata, const char *path,
                    struct fuse_file_info *fi);
```

[*libfuse* reference: fuse.h#L389-L437]

This function is called to `open()` a file. As mentioned above, `fi->fh` must be filled in with a file handle to identify the file that has been opened. `watdfs_cli_open` should return 0 on success or `-errno` if an error occurs. Note that this return value is different from the `open()` system call that returns a file descriptor or -1.

We will use only the following flags during `open`: `O_CREAT`, `O_APPEND`, `O_EXCL`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR`.

Important: The `fuse_file_info` structure contains useful information such as the flags that the file is requested to be opened with.

Hint (P2): You can tell whether a file is requested to be opened *read only*, *write only*, or *read/write* by performing a *bitwise-and* (&) of the flags and `O_ACCMODE` (refer to libc access modes).

3.4.3 watdfs_cli_release

```
int watdfs_cli_release(void *userdata, const char *path,
                       struct fuse_file_info *fi);
```

[*libfuse* reference: fuse.h#L492-L504]

This function is called on a `close` system call. However, `watdfs_cli_release` is performed asynchronously; that is, the `close` system call may complete and return to the application before the actual execution of `watdfs_cli_release` completes.[7] There is exactly one `watdfs_cli_release` per `open`, with the same file name, flags, and file handle (`fi->fh`).

3.5 Reading and Writing

The following calls correspond to systems calls that support reading and writing data.

3.5.1 watdfs_cli_read

```
int watdfs_cli_read(void *userdata, const char *path, char *buf,
                    size_t size, off_t offset,
                    struct fuse_file_info *fi);
```

[*libfuse* reference: fuse.h#L439-L448]

This function reads into `buf` at most `size` bytes from the specified `offset` of the file. It should return the number of bytes requested to be read, except on `EOF` (return the number of bytes actually read) or error (return `-errno`).

[7] You can ignore this behavior in P1 but you must handle it in P2.

Hint: The amount of data requested to be read may be larger then the maximum amount of data that can be used as an array argument in our RPC library (Section 4.2). You **must** handle this case within the same `watdfs_cli_read` call.

Important: You should ensure that the buffers used in the RPC calls are of exact sizes, not over or under provisioned.

3.5.2 `watdfs_cli_write`

```
int watdfs_cli_write(void *userdata, const char *path,
                     const char *buf, size_t size, off_t offset,
                     struct fuse_file_info *fi);
```

[*libfuse* reference: fuse.h#L451-L460]

This function writes `size` number of bytes from `buf` into the file at the specified `offset`. It should return the number of bytes requested to be written, except on error (`-errno`).

Hint: The amount of data requested to be written may be larger than the maximum amount of data that can be used as an array argument in our RPC library (see Section 4.2). You **must** handle this case within the same `watdfs_cli_write` call.

3.5.3 `watdfs_cli_truncate`

```
int watdfs_cli_truncate(void *userdata, const char *path,
                        off_t newsize);
```

[*libfuse* reference: fuse.h#L379-L387]

This function changes the size of the file to `newsize`. If the file previously was larger than this size, the extra data is deleted. If the file previously was shorter, it is extended, and the extended part is filled in with null bytes ('\0').

Important: `truncate` can be called without opening the file and should succeed if the file has the `write` permissions.

3.5.4 `watdfs_cli_fsync`

```
int watdfs_cli_fsync(void *userdata, const char *path,
                     struct fuse_file_info *fi);
```

[*libfuse* reference: fuse.h#L506-L511]

This function should flush the file data specified by `path` and `fi`.

3.6 Changing Metadata

3.6.1 `watdfs_cli_utimensat`

```
int watdfs_cli_utimensat(void *userdata, const char *path,
                         const struct timespec ts[2]);
```

[*libfuse* reference: fuse.h#L638-L650]

This will change the file access and modification time with nanosecond resolution.

Important: There are different semantics based on the `ts` argument; you should **carefully** read the documentation of utimensat.

4 RPC Library Documentation

We provide an RPC library for use in client and server communication. The RPC library supports making remote procedure calls from a client to a server. To use this library, function handlers and their expected types must be defined (Section 4.1) and registered (Section 4.2) in the WatDFS server before it starts accepting connections.

4.1 RPC Protocol Definitions

The RPC calls that you should implement to satisfy the client requests are listed below with their argument types. Each RPC requires an integer (`retcode`) as an output argument, which is usually `0` or `-errno`, except for `read` and `write` calls, where it is the number of read/written bytes or `-errno`. See Section 3.1 for more details.

You must **exactly** follow this protocol specification, as the automated tests will verify it.

Important: Rather than serializing each argument in a structure `T` (e.g. `fuse_file_info`), you should simply serialize the entire structure as a character array of `sizeof(T)`.

4.1.1 getattr

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
statbuf	no	yes	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

4.1.2 mknod

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
mode	yes	no	no	ARG_INT
dev	yes	no	no	ARG_LONG
retcode	no	yes	no	ARG_INT

4.1.3 open

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	yes	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Important: The retcode for the `open` RPC call should be 0 or `-errno`. This return value matches the expected return value of `watdfs_cli_open`, but is different from the system call `open`, which returns a file descriptor.

4.1.4 release

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

4.1.5 read

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
buf	no	yes	yes	ARG_CHAR
size	yes	no	no	ARG_LONG
offset	yes	no	no	ARG_LONG
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

4.1.6 write

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
buf	yes	no	yes	ARG_CHAR
size	yes	no	no	ARG_LONG
offset	yes	no	no	ARG_LONG
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

4.1.7 truncate

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
newsize	yes	no	no	ARG_LONG
retcode	no	yes	no	ARG_INT

4.1.8 fsync

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
fi	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

4.1.9 utimensat

label	is_input	is_output	is_array	type
path	yes	no	yes	ARG_CHAR
ts	yes	no	yes	ARG_CHAR
retcode	no	yes	no	ARG_INT

Keep the above tables handy when implementing the RPC calls. Be extra careful with the data types and settings for each RPC call, as they can be easily mixed up.

4.2 Registering RPC calls on the server with rpcRegister

Function calls must be registered using `rpcRegister` before they can be used by remote clients. This function tells the RPC library which function to call at the server when a client executes an `rpcCall` with a specified name and arguments.

```
int rpcRegister(char *name, int *argTypes, skeleton f);
// skeleton is defined as:
typedef int (*skeleton)(int *, void **);
```

Here, `skeleton f` is the address of a server function that is mapped to a given rpc `name` by the server. Skeletons return an integer error code indicating success with a zero or failure with a negative error code. The inputs and outputs of the skeleton functions are provided as arguments to the skeleton and marked as inputs or outputs respectively using the separate `argTypes` array (see below).

`rpcRegister` will return an integer indicating success (0) or failure (a negative number).

The `argTypes` array specifies the types of the argument, and whether the argument is an input, output, or both for the function call. Each argument has an integer to encode the type information (refer to the tables described in Section 4.1), which will collectively form the `argTypes` array. The `args` array is an array of pointers to the different arguments, which will be discussed in the next section. Thus, `argTypes[0]` specifies the type information for `args[0]`, and so on.

Since it is not known how many arguments there are, the last value in the `argTypes` array must be 0. Consequently, the size of `argTypes` is 1 greater than the size of `args`. There is no restriction on the number of input and output arguments placed in the `args` vector or how they are ordered. However, the `argTypes` array must correctly correlate with the `args` vector.

The `argTypes` integer (of size 4 bytes) consists of the following sub-components:

1. The first byte will specify the input/output nature of the argument. Specifically, if the first bit is set then the argument is input to the server. If the second bit is set the argument is output from the server. The third bit indicates whether the argument is an array type. The remaining 5 bits of this byte are currently undefined and must be set to 0. For convenience, the following definitions are provided:

```
#define ARG_INPUT 31
#define ARG_OUTPUT 30
#define ARG_ARRAY 29
```

2. The second byte contains argument type information. The following definitions can be used to describe the different types.

```
#define ARG_CHAR 1
#define ARG_SHORT 2
#define ARG_INT 3
#define ARG_LONG 4
#define ARG_DOUBLE 5
#define ARG_FLOAT 6
```

3. The last two bytes of the argument type integer specify the length of the array. Hence, arrays are limited to a length of $2^{16} - 1$ (defined as `MAX_ARRAY_LEN`). If the argument is not an array, then these last two bytes must be 0. It is expected that the client programmer will have reserved sufficient space for the buffers used for any input/output of type arrays.

Example: To indicate an input array of 20 integers, you can set the `argType` to:
`(1 << ARG_INPUT) | (1 << ARG_ARRAY) | (ARG_INT << 16) | 20`.

Note: When a function is registered with `rpcRegister`, the server does not know the length of the array that the client will set; therefore, on the server side, you must specify the length of the array as 1 in the call to `rpcRegister`.

We show a full example of registering a `sum` function that calculates the sum of all numbers in an int array:

```
#define ARG_COUNT 2 // Number of RPC arguments
#define LENGTH 1u // It is an array

// Create the argTypes array. Note the extra length (ARG_COUNT+1),
// which is for the last 0 to indicate the end.
int argTypes[ARG_COUNT+1];
// First type is the output sum of type int.
argTypes[0] = (1u << ARG_OUTPUT) | (ARG_INT << 16u);
// Second type is the input of int arrays.
argTypes[1] = (1u << ARG_INPUT) | (1u << ARG_ARRAY) |
              (ARG_INT << 16u) | LENGTH;
// 0 to indicate end of arguments.
argTypes[2] = 0;

// Finally register the function with the RPC library. This indicates
// that when client makes a "sum" call with given argTypes, sumFunction
// should be executed.
rpcRegister("sum", argTypes, sumFunction);
```

```
int sumFunction(int *argTypes, void** args) {
    // The incoming arguments are an output int and an input int array.

    // Get the lowest 2 bytes to find the array length.
    int len = argTypes[1] & ((1 << 16) - 1);

    // Save the result to the output arg (which is of type int)
    int *total = (int *) args[0];

    // Cast to get the input array of ints.
    int *to_sum = (int *) args[1];

    for (int pos = 0; pos < len; pos++) {
        *total += to_sum[pos];
    }

    return 0; // Success.
}
```

Important: The RPC server may call your registered functions from any thread that it has available, so you should not keep or use any thread-local state, as it will not be available on subsequent (future) function executions!

4.3 Executing RPC calls on the client with rpcCall

The client executes an RPC by calling the `rpcCall` function.:

```
int rpcCall(char *name, int *argTypes, void **args);
```

The return value after calling the function is the result of executing the rpcCall itself, and **not the result of the procedure** that the rpcCall was executing. That is, if the rpcCall failed, that would be indicated by a negative number and success is indicated by a 0. It is expected that you check the return value for errors. If your function is expected to return a value (e.g., the total sum), it should be specified as an output argument in the same way as defined in `rpcRegister`.

The `name` argument is the name of the remote procedure to be executed, a procedure that matches the name with the same `argTypes` must have been registered at the server. If no such procedure exists, an error code will be returned indicating so.

If a client wants to use the `sum` method that we registered in Section 4.2 to calculate: `result = sum(int vect[LENGTH])`, the code would be:

```
// The number of RPC arguments is same as defined in `rpcRegister`.
#define ARG_COUNT 2
// We are on the client, so we know the length of the input array.
#define LENGTH 23u
```

```
// Create `argTypes` the same way as in `rpcRegister`.
int argTypes[ARG_COUNT+1];
// First argument is the output int to store the final sum.
argTypes[0] = (1u << ARG_OUTPUT) | (ARG_INT << 16u)
// Second argument is the input int array of length LENGTH.
argTypes[1] = (1u << ARG_INPUT) | (1u << ARG_ARRAY) |
              (ARG_INT << 16u) | LENGTH;
argTypes[2] = 0; // Terminator

int result;
int vector[LENGTH]; // Assume this is intialized with values.

// Create the `args` array, which is a array of `void` pointers.
void **args = (void **)malloc(ARG_COUNT * sizeof(void *));
// Recall, 1st argument is the output, which will be stored in `result`.
args[0] = (void *)&result;
// Second argument is the input `int` array.
args[1] = (void *)vector;

// Make the RPC call by using the registered "sum" RPC call.
int ret = rpcCall("sum", argTypes, args);

if ret < 0 {
    // Handle `rpcCall` error.
} else {
    // Success. The sum will be available in `result`.
}

free(args); // Clean up args.
```

4.4 Other RPC library calls

Four other calls must be made to use RPC library. All these calls return `0` on success and a negative number on failure.

4.4.1 rpcServerInit

Before registering any functions the server must first call `rpcServerInit`, this will initialize any server state that the RPC library must maintain.

4.4.2 rpcExecute

When the server is ready to serve calls, it should call `rpcExecute`. This will transfer control to the RPC library, and when an RPC call is received the registered functions will be called. `rpcServerInit` must be called before `rpcExecute` is called. When `rpcExecute` is called the server will at the end of initialization print the address and port that the server is listening

on, which must be set by the client, as described in `rpcClientInit`.

```
export SERVER_ADDRESS={address}
export SERVER_PORT={port}
```

4.4.3 rpcClientInit

The client should call `rpcClientInit` to initialize a connection with the server. `rpcClientInit` should be called before any `rpcCalls` are made. To connect with the server, the environment variables `SERVER_ADDRESS` and `SERVER_PORT` must be set before the client binary is executed. For example:

```
$ export SERVER_ADDRESS={address}
$ export SERVER_PORT={port}
$ ./watdfs_client {args}
```

4.4.4 rpcClientDestroy

The client should call `rpcClientDestroy` when they are finished interacting with the server. This will terminate connections with the server.

4.4.5 RPC Library Tips

The RPC library logs each call, the argument types, and the return value in `rpc_client.log` and `rpc_server.log` respectively. Consult these logs if your RPC calls do not succeed.

Important: All rpc calls return values to indicate success or error. Ensure that you are checking these values and handling error cases. For example, you should stop execution if `rpcServerInit` or `rpcClientInit` fail.

5 Other Resources

5.1 Resources for working with FUSE

The FUSE headers are available online: libfuse/include/fuse.h

A general tutorial for the FUSE functions are available online at: cs.nmsu.edu/fuse-tutorial

Some pointers on the purpose of each FUSE call: cs.hmc.edu/fuse/fuse_doc.html

5.2 Resources for system calls

While completing these projects, you will need to make many system calls. To understand their functionality, you should consult the man pages. Some system calls you might need are listed in the table below.

A general tutorial on how to use file system calls is available at: eecs.utk.edu/Syscall-Intro.

5.3 General Resources

For those who are not familiar with the process of creating a static library, consult the following link: http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html

System Calls	Description	Links
open, creat	Opening a file	open.2.html
close	Closing a file	close.2.html
mknod	Creating a file	mknod.2.html
read/write	Reading and writing to a file	read.2.html
pread, pwrite	Reading and writing a file to offsets	pread.2.html
stat, lstat, fstat	Getting file attributes	fstat.2.html
truncate ftruncate	Truncating a file to a length	truncate.2.html
fsync	Sync a file to storage	fsync.2.html
utimensat	Change file modified/access times	utimensat.2.html

In coding this Project, you may find helpful to use a debugger, e.g., gdb is available on the CSCF environment; a gdb tutorial is available at: https://beej.us/guide/bggdb/

POSIX threads (pthreads) are a standardized interface for threads on UNIX systems and a great tutorial with code examples can be found here: https://hpc-tutorials.llnl.gov/posix/

A more general tutorial on threads including details of different threading interfaces is given in: http://www.cs.cf.ac.uk/Dave/C/node29.html

6 Project 1

6.1 Remote Access Model

In the remote access model, every request is forwarded from the client to the server. A remote file is never stored at the client.

To implement the remote access model in WatDFS, you will transform each of the described FUSE functions (Section 3) into an RPC call from the client to the server. The server will transform the RPC call into a system call, execute the operation on the server directory, and return the result to the client. The server should perform all operations on the files using appropriate system calls and should not buffer writes or cache files in memory. That is, if you receive a `write` call, the write should be performed on the remote file system and should not be performed by keeping a copy of the file in local memory and writing to it.

In P1, the client and server should be stateless as all the information needed to perform the required operation is provided by the calling function.

The remote access model will serve as a warm up to familiarize you with FUSE, RPC calls, and the required system calls.

6.2 Requirements

You are required to implement WatDFS using the Remote Access Model as described in this specification (Section 6.1). In particular, you are required to implement the client and server components. You must use C++ to implement your solution.

Your `Makefile` should compile and produce the following:

- **libwatdfs.a**: A library containing the implementation of your client side library. You **should not have a main function** in your library; we provide it in `libwatdfsmain.a`.
- **watdfs_server**: An executable of your server.
  - Your server executable should have a `main` function.
  - Your server must parse the command line arguments to read the directory where the server will persist data (see Section 6.3).
  - Your server should register all required RPC functions.

We will provide:

- **rpc.h and watdfs_client.h**. You **must not change** these headers.
- **librpc.a**, which contains the RPC library.
- **libwatdfsmain.a**, which implements the client `main()` and sets up FUSE.

We will also provide a starter code to help you get started with the Project:

- **watdfs_client.c**: with function definitions for the required methods, and some comments.
- **watdfs_server.c**: with a starter main method, and some comments
- **Makefile**, that will compile the starter code.

Note: You can reorganize the provided code if you want, as long as the `Makefile` generates the required output described above.

6.3 Testing

To compile your code:

```
# Compile libwatdfs.a, watdfs_server, and watdfs_client
$ make all
# Compile after cleaning
$ make clean all
```

To run your server (in terminal 1 on e.g., `ubuntu1804-008`):

```
ubuntu1804-008$ mkdir -p /tmp/$USER/server
ubuntu1804-008$ ./watdfs_server /tmp/$USER/server
[...initialization logs...]
export SERVER_ADDRESS=ubuntu1804-008
export SERVER_PORT=12345
# If you want to avoid printing the provided code log messages to the cli:
ubuntu1804-008$ ./watdfs_server /tmp/$USER/server 2>/dev/null
export SERVER_ADDRESS=ubuntu1804-008
export SERVER_PORT=12345
```

Here, `/tmp/$USER/server` is the canonical central directory of WatDFS, where the server will perform all file operations, including creating and saving files.

If the server implementation is correct, it will print the `SERVER_ADDRESS` and `SERVER_PORT` it is running at, as shown above.

Important: This directory **must not be** located in your home directory when you are the testing on the UW student environment servers. Home directories on those servers are themselves remotely mounted, and FUSE exhibits strange behavior if run on top of such mounted directories. Instead, use a directory that is local to the machine, such as `/tmp/$USER/server` shown above.

To run your client (in terminal 2 on e.g., `ubuntu1804-004`):

```
# Export the host and port where the server is running. This should be
# taken from the server output.
ubuntu1804-004$ export SERVER_ADDRESS=ubuntu1804-008
ubuntu1804-004$ export SERVER_PORT=12345
ubuntu1804-004$ mkdir -p /tmp/$USER/cache /tmp/$USER/mount
ubuntu1804-004$ ./watdfs_client -s -f -o direct_io /tmp/$USER/cache \
/tmp/$USER/mount
# Append "2>/dev/null" to the above command if you want to prevent
# printing all the provided code logs to the cli.
```

Here, `/tmp/$USER/cache` is the directory where you will cache files in P2 (you will not use this in P1). It is passed as the `path_to_cache` argument in `watdfs_cli_init`.

`/tmp/$USER/mount` is the directory that you'll use to interact with your WatDFS implementation. Your WatDFS FUSE client implementation is *mounted* on top of this directory and thus forwards any file operation (e.g., `open()`) performed inside this directory to a client function (e.g., `watdfs_cli_open()`) that you've implemented.

Important: As with the server, make sure you are not using your home directory as `cache` and `mount` when testing on the student environment.

You can test your client with bash commands (in terminal 3 on the *same server* as `watdfs_client`, e.g., `ubuntu1804-004` above):

```
# Create an empty file.
ubuntu1804-004$ touch /tmp/$USER/mount/myfile.txt
# Write to a file.
ubuntu1804-004$ echo "CS454 is fun" > /tmp/$USER/mount/myfile.txt
# Read file. Should print "CS454 is fun".
ubuntu1804-004$ cat /tmp/$USER/mount/myfile.txt
# Get file attributes.
ubuntu1804-004$ stat /tmp/$USER/mount/myfile.txt
```

Important: FUSE can run a client multi-threaded. However, we will always run your client single threaded (`-s`) and with the `-o direct_io` flag to disable kernel caching. Run `./watdfs_client` (no args) to see more options. For example, without the (`-f`) flag, the client will run in the background.

Important: If you cannot start your client and get a mount point error, execute the following

to manually unmount an already mounted directory.

```
$ fusermount -u /tmp/$USER/mount
```

Note: You can run multiple clients that connect to the same server. To start a second client that connects to the server started above, you can execute (in terminal 4 on e.g., `ubuntu1804-002`):

```
ubuntu1804-002$ export SERVER_ADDRESS=ubuntu1804-008
ubuntu1804-002$ export SERVER_PORT=12345
ubuntu1804-002$ mkdir -p /tmp/$USER/cache /tmp/$USER/mount
ubuntu1804-002$ ./watdfs_client -s -f -o direct_io /tmp/$USER/cache \
/tmp/$USER/mount
```

Important: If you're running both the clients on the same machine, you have to make sure that the cache and mount directories are different for both the clients, but that the `SERVER_ADDRESS` and `SERVER_PORT` are the same.

6.4 Submission

You should submit your **Project** on Marmoset as a single zip file for automated testing.[8] Your **Project** zip should contain your code and the `Makefile`.

Marmoset's build scripts require that your Makefile be titled `Makefile` and that it is placed in the root directory of your submission zip file. If the `Makefile` is not located in the right location, you will get a compilation error indicating that the Makefile could not be found—**make sure that when you unzip your code, the `Makefile` is not in a subfolder!**

```
$ ls
watdfs_client.cpp
watdfs_client.h
watdfs_server.cpp
rpc.h
debug.h
Makefile
$ make zip
 adding: watdfs_client.cpp (deflated 17%)
...
# Submit watdfs.zip to Marmoset
```

More details about Marmoset:

If your submitted program does not compile or run successfully on its own, your submission will receive a result of "did not compile" and the detailed test results will contain something similar to the error message you get if you ran your program yourself. In this case, your submission will not be tested with any of the tests.

[8] You can also try using the `marmoset_submit` command.

If your program runs successfully on its own, it will be tested with all of the public tests. If it fails any of the public tests, the detailed test results will display an error message for that public test. In this case, your submission will not be tested with any of the release tests.

If it passes all of the public tests, you will have the option to see the results of (a subset of) the release tests. If you do so, you will use up one of your `release tokens` for that Project. For both the projects, you will be initially given 5 release tokens. Used tokens will be regenerate every 24 hours. If the deadline expires before your token regenerates, you can still submit, but you will not be able to tell how your submission did on the release tests.

If your submission fails a release test and you use a token to see the results, you will see the detailed test results of 2 release tests. If your submission passes all the release tests, you will not see any release tests in the detailed test results. If you fail a release test, please do not attempt to guess what that test case might be and **do not ask or speculate about the test cases on Piazza. The correct action when failing a release test is to re-examine your own test suite and to update it to reproduce the error.**

If you have developed a test that resembles the functionality of the release test, cannot reproduce the error, and are stuck, you may make a private Piazza post. You must include:

- Your Marmoset submission number that contains the current version of your code
- The Marmoset test case you are failing
- The code for a test case that you have developed to reproduce the Marmoset test case
- The logs from your test case
- What you expected the behaviour of the test to be, and why

A TA will try to help you. **Do not abuse this feature, e.g., by making many such posts or providing low-effort t est c ases, a s o therwise T As m ay n ot respond. Remember that there are many students in the course and TAs cannot be on-call 24/7. Importantly, responses to questions take time (particularly as Project deadlines near).** Questions will be answered on a first-come, first-served, best-effort basis.

Hint: `recv()` socket calls used in the RPC library will hang if they anticipate receiving more data than the other side sends. If a Marmoset test "times out", it is quite likely that you are either not following the protocol described in the Project or you have used the RPC library incorrectly. Make sure that you are setting the length of arguments correctly on both the client and server!

6.5 Evaluation

We will evaluate your system for the following:

1. Your code compiles on `linux.student.cs.uwaterloo.ca`.
2. Has the following functionality (we may add or remove particular tests):
   (a) Opening or creating a file. We will only use the following flags: `O_CREAT`, `O_APPEND`, `O_EXCL`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR`.
   (b) Writing to a file
   (c) Reading from a file
   (d) Getting file attributes and updating file modification times

(e) Synchronizing a file
(f) Truncating a file
(g) Closing a file
(h) Using appropriate error codes for different error scenarios

3. Your code follows the specified protocol for RPC calls.

6.6 Tips

You should implement functions in the following order: `getattr`, `mknod`, `open`, and `release`. These will allow you to support opening and creating a file. Next, you should implement: `write`, `read`, and `truncate`. These functions will allow you to read and write data to a file. Finally implement: `utimensat` and `fsync`.

To assist you in debugging the calls made to your implemented functions, we log before your functions are called and log the return value of your calls. The logfile is stored as `watdfs.log` in the directory where you start your client.

Recall from Section 3.1 that *libfuse* expects functions to return 0 or `-errno`. However, most system calls (e.g. `stat`) return 0 or -1, and set `errno`. If a system call returns -1, in your WatDFS server you should set `retcode` in the RPC call to `-errno` and in the WatDFS client return that error from your function to *libfuse*.

6.6.1 Handling failures

Ensure that you are checking the return values of all function calls you make. A non-zero return *usually* indicates an error (except for the `read` and `write` calls as described in Section 3.5). You should detect errors just after the function call that caused it and return the error code appropriately. **Failing to check the return code of functions is a common source of bugs**.

6.6.2 Assumptions about the system

You can assume that the server and client will not fail or crash during normal operations. You can also assume that both the client and server will have the appropriate directory permissions to create and modify files. In general, the errors that you need to handle are system call and rpc call failures.

6.6.3 Specific Do's and Don'ts

- Do not use `strlen(buffer)` to calculate the length of buffers. The buffers used in read and write calls are not char arrays representing strings. You should assume that buffers contain arbitrary binary content that is not delimited by '\0'.
- Do add small comments, for example to remind you what you did in P1 that might be helpful for P2.
- Do make good use of functions (and macros), instead of giving in to the temptation of copy pasting code.
- Don't change the WatDFS client interface. We will be using these interfaces to run the FUSE client. In particular, do not modify `rpc.h` or `watdfs_client.h`.
- Do test that your code compiles and runs on the `linux.student.cs.uwaterloo.ca` servers before submitting to Marmoset.

- **Do not** use any public or web-based source code repository hosting service other than https://git.uwaterloo.ca.

7 Project 2

7.1 Upload Download Model

In Project 2, you will implement WatDFS using the upload-download model. In this model, instead of directly accessing the remote files by forwarding the system calls to the server as done in A1, files are **copied** from the server to the client and the client performs all operations locally. In this Project, you will use a timeout-based caching method and mutual exclusion of writes to ensure consistency.

In Project 2, you will build on your Project 1 solution and implement new RPC calls to support the upload-download model. You have to design the protocol for these calls yourself, similar to the ones described in Section 4.1, although we make some suggestions below (Section 7.2.4). Consequently, we will test for end-to-end system correctness.

You should review Sections 3 and 4 to remind yourself of the FUSE operations and the RPC library functions.

7.1.1 Performing operations locally

The upload-download model requires performing all operations locally, which requires transferring the file from the server to the client. To satisfy this requirement, a `watdfs_cli_open` call should always start by trying to transfer the file from the server to the client.

Hint: You must handle cases where the file exists or does not exist on the server. The correct action will depend on the flags passed to open (such as `O_CREAT`).

7.1.2 Mutual exclusion on the server

To ensure mutual exclusion of writes, the server should return an error (`-EACCES`) if a client attempts to `open` a file (with write mode) and the file is already opened (in write mode). The file can be opened again by another client once the file has been closed (`watdfs_cli_release`). Any number of concurrent readers (from multiple FUSE instances) to the same file are allowed.

In short, multiple WatDFS clients can read a file concurrently but only one WatDFS client can write to the file.

7.1.3 Mutual exclusion on the client

On the client side, it is prohibited for an application to `open` the same file multiple times. If an application attempts to `open` a file that is already open, the WatDFS client should return `-EMFILE`. This error has precedence over `-EACCES`, i.e., the `-EMFILE` error condition should be checked first.

7.1.4 Atomic file transfers

Copying files to and from the server must be performed atomically. For example, if the file `F` is being written to the server by client `C1`, and `F` is being read from the server by a second

client C2, then C2 should see all or none of the updates made to F by C1.

7.1.5 Timeout-based caching

You will implement a variant of NFS caching semantics using timeouts. Under this caching scheme, a cached file may be used to satisfy a read request if it satisfies the freshness condition at time T for some freshness interval t.

Let Tc be the time the cache entry was last validated by the client. T_client represents the time that the file was last modified as recorded by the client. T_server represents the time that the file was last modified as recorded by the server.

read(): The freshness condition at time T for a file that is opened for *only reads* (O_RDONLY) is defined as follows. If at time T: (i) [(T − Tc) < t] or (ii) [T_client == T_server], then the file can be served from the locally cached file copy. Otherwise, a new copy of the file must be retrieved from the server and cached at the client. Note that T − Tc < t should **not** be determined by accessing the server. Conversely, T_client == T_server should be determined by accessing the server.

If the first condition check fails, you'll need to determine T_server by consulting the server. If T_server is retrieved and is equal to T_client then Tc should be updated to the current time. Otherwise, the file has changed at the server and a fresh copy should be fetched from the server to the client. After fetching, you should update T_client to be equal to T_server, and update Tc to the current time.

write(): Writes by a client should be applied to the local copy of the file, and periodically written back to the server. You will do this by checking the freshness condition for writes at the end of a write/truncate call. If at that time T, [(T − Tc) < t] or [T_client == T_server], then you can return immediately. Otherwise, you must (synchronously) write client's copy of the file back to the server and update T_server to T_client. When the client is done with the file, as indicated by watdfs_cli_release, the file should always be written back to the server.

fsync(): If a client application issues fsync, then the client's copy of the file must be written to the server immediately and T_server and Tc should be updated. If the file is opened in read only mode, return an error.

Note: On watdfs_cli_release, the client copy of the file should be closed (close system call), but it should remain in the cache directory. The cached copy of the file should not be deleted.

7.1.6 Caching for other calls

Consider the following calls divided into 2 groups:

- Read calls: watdfs_cli_getattr
- Write calls: watdfs_cli_truncate, watdfs_cli_mknod, watdfs_cli_utimensat (setting the file *modified* time)

The cache behavior of these calls depends on the *open* state of the file at the client.

If the file has not been opened: You should try to open and transfer the file from the

server, perform the operation locally, transfer the file back to the server (for write calls), and close the file.

If the file is already open: Regular read or write freshness checks should take place. There are 2 cases:

- The file is open in read only mode: Only read calls are allowed and should perform freshness checks before reads, as usual. Write calls should fail and return `-EMFILE`.
- The file is open in write mode: Read calls should not perform freshness checks, as there would be no updates on the server due to write exclusion and this prevents overwriting local file updates if freshness condition has expired. Write calls should perform the freshness checks at the end of writes, as usual.

7.1.7 Client side cache

Recall the definition of `watdfs_cli_init` from Section 3.2.1:

```
void *watdfs_cli_init(struct fuse_conn_info *conn,
                      const char *path_to_cache, time_t cache_interval
                      int *ret_code);
```

The client should persist data (cached copies of files) in the `path_to_cache` directory. All client operations should be performed on its copy of the file, using the local file system. The client should not store its local copies of files in memory. That is, when a client performs a write, make a `write` system call against the client's copy of the file on disk—simply writing to an in-memory buffer is incorrect!

The cache interval `t`, in seconds, is passed as `cache_interval`. `cache_interval` is determined by setting the environment variable `CACHE_INTERVAL_SEC` before executing the client (see Section 7.5 below).

7.1.8 Server side

As in the remote access model, the server should perform operations on the file system and not buffer writes or cache the file in memory.

7.2 Suggested Implementation Strategy

We recommend implementing the requirements in the following order: build the file upload/download mechanism (see Section 7.2.2), enforce mutual exclusion of writes (see Section 7.2.3), and enforce timeout caching (see Section 7.1.5). The rest of this secion provides more specific suggestions for implementing the upload-download model in WatDFS. You do not have to follow these suggestion as long as you adhere to defined API. However, we recommend that you understand this method before developing your own.

7.2.1 Opening a file

When a file is opened, it is opened with a file access mode (i.e. `O_RDONLY`, `O_WRONLY`, `O_RDWR`). However, since files opened on the server will be read and files opened on the client will be written to while creating local cached copies, these modes cannot be passed directly to `open` on the client or server.

When `watdfs_cli_open` is called, you should (try to) copy the file from the server to the client so the client can apply operations locally. Therefore, `open` should be called at both the client and the server as part of caching the file locally, resulting in two different file descriptors which you should track at the client. As part of these `open` calls, you should satisfy the mutual exclusion requirements (suggestions in Section 7.2.3).

Once the file has been copied to the client, the original flags received by `watdfs_cli_open` must be used, such that the file handle returned from this call respects the flag properties (e.g., read-only, write-only).

Opening a file should also initialize metadata at the client that is needed to check the freshness condition for the file (`Tc`). You can use the file modification time of the file to track `T_client` and `T_server`.

7.2.2 Steps for transferring the file from the server to the client

To transfer a file from the server to the client you should: truncate the file at the client, get file attributes from the server, read the file from the server, write the file to the client, and then update the file metadata at the client. You should reuse RPC calls from Project 1 to complete this task.

7.2.3 Enforcing mutual exclusion

To ensure that a file has only a single `open` writer at a time, the server should keep track of open files. The server should maintain a *thread synchronized* data structure that maps filenames to their status (open for write, open for read, etc.). If the server receives an `open` request for write to a file that has already been opened in write mode, the server will use this data structure to discover the conflict and return `-EACCES`. When the server receives a message to close a file it has opened in write mode, the data structure should be modified to indicate that the file is now available to a writer.

To ensure that the same client does not open the same file multiple times, you must track current open files and their modes (read-only or write) at the client. You should use your already tracked metadata to determine if the file has already been opened; if it has return `-EMFILE`.

7.2.4 Ensuring file transfers are atomic

To ensure that file transfers are atomic, you should implement two RPC calls to indicate that a file is in transfer either to or from the server. A file can be transferred from a server to multiple clients in parallel. However, writing back the file to the server should be mutually exclusive to these reads. To achieve these atomic transfers you should mark a file as in transfer for reads or in transfer for writes.

To assist you with this task, we have provided an implementation of reader-writer locks, in the files: `rw_lock.h` and `rw_lock.c`, which define the struct `rw_lock_t`, and methods to acquire and release locks in read or write mode.

The API of the `rw_lock_t` is similar to that of `pthread_mutex_t`, and is defined as follows (more details are in `rw_lock.h`). All functions return `0` on success and `-errno` (< 0) on failure.

To setup and tear down the lock, use the following functions:

```
int rw_lock_init(rw_lock_t* lock); // Initialize the lock.
int rw_lock_destroy(rw_lock_t* lock); // Destroy the lock.
```

To acquire and release the lock, use the following functions:

```
// Acquire the lock in mode (RW_READ_LOCK, RW_WRITE_LOCK)
int rw_lock_lock(rw_lock_t* lock, rw_lock_mode_t mode);
// Release the lock from mode (RW_READ_LOCK, RW_WRITE_LOCK),
// you must have been an owner of the lock to release it
int rw_lock_unlock(rw_lock_t* lock, rw_lock_mode_t mode);
```

Given the reader-writer lock definition, the following two RPC calls should be implemented:

1. `lock(path, mode)`
2. `unlock(path, mode)`

These calls lock and unlock the file (located at `path`) for transfer using the read or write `lock_mode`, respectively.

7.2.5 Releasing a file

If the file was opened in write mode, the file should be flushed from the client to the server (described below). The file should then be closed at both the client and the server. The release RPC call from Project 1 can be modified to incorporate the mutual exclusion protocol. The file metadata that is tracked at the client should also be released.

7.2.6 The Effects of Asynchronous release

Recall from the description of the *libfuse* functions that `watdfs_cli_release` is called asynchronously after a file is closed. Specifically, when `close` is called on a file, *libfuse* will call `watdfs_cli_release`, but will not wait for `watdfs_cli_release` to return before `close` returns. However, because `watdfs_cli_release` is responsible for transferring a writable file from the client to server and unlocking it, subsequent `open`/`close` calls to the same file may not succeed because `watdfs_cli_release` has not completed. It is for this reason that we require that if release has not yet completed on the same file, `open`s should fail with `EMFILE`. The automated tests will look for this and loop over `open()` with `EMFILE` to open a file that has just been closed. When you are writing your own test applications, you should look at the `errno` when `open` calls fail on subsequent `open`/`close` operations, and if it is `EMFILE`, retry the `open` operation.[9]

7.3 System Manual (15 marks)

For Project 2, you must submit a system manual (as a PDF document, around 3-4 pages). It should include the following items:

1. You should discuss your design choices so that we can understand how key functionality

[9]This looping behavior should only be done in your testing applications, and not in your *libfuse* code! Since we are running FUSE in single-threaded mode, looping in your library code on `EMFILE` will result in a deadlock!

was implemented. These include all the steps you take to copy a file from the client to server and vice-versa, how the transfers are atomic, all the steps to test for cache invalidation and how you ensure mutual exclusion at client and server. Each of these functionalities' discussions are marked separately; you should complete all the sections.
2. Clearly identify the functionalities of the Project that has not been implemented and how you'd finish it. **You will receive marks based on these descriptions, even if your code is incomplete.**
3. List the error codes that you returned (outside of error codes returned from system calls directly)
4. You should discuss (with steps) at least 2 different ways in which you tested your Project (e.g., describe a series of system calls that you have made to ensure your Project works)

7.4 Requirements

You will implement WatDFS using the upload/download model as described in this specification. In particular, you will implement the client library and the server. The code and `Makefile` requirements for submission of Project 2 are the same as Project 1 (Section 6.2).

In addition, you must also submit your system manual (Section 7.3), which must be named `MANUAL.pdf`.

In addition to the files provided in Project 1, we will provide:

- `rw_lock.h` and `rw_lock.c`: These contain an implementation of a reader-writer lock, as defined in Section 7.2.4.

7.5 Testing

To test your client and server, we will run the commands listed in Section 6.3, but will modify the execution of the client as follows:

Compile your code:

```
# Compile libwatdfs.a, watdfs_server, and watdfs_client
$ make all
# Compile after cleaning
$ make clean all
```

Start your server (in terminal 1 on e.g., `ubuntu1804-008`):

```
ubuntu1804-008$ mkdir -p /tmp/$USER/server
ubuntu1804-008$ ./watdfs_server /tmp/$USER/server
export SERVER_ADDRESS=ubuntu1804-008
export SERVER_PORT=12345
```

Start your client (in terminal 2 on e.g., `ubuntu1804-004`):

```
# Export the host and port where the server is running. This should be
```

```
# taken from the server output.
ubuntu1804-004$ export SERVER_ADDRESS=ubuntu1804-008
ubuntu1804-004$ export SERVER_PORT=12345
# New for P2:
ubuntu1804-004$ export CACHE_INTERVAL_SEC=5 # or any integer value >= 0
ubuntu1804-004$ mkdir -p /tmp/$USER/cache /tmp/$USER/mount
ubuntu1804-004$ ./watdfs_client -s -f -o direct_io /tmp/$USER/cache \
/tmp/$USER/mount
```

7.6 Submission

You should submit your **Project** on Marmoset as a single zip file for automated testing. Your **Project** zip should contain your code and the `Makefile`. Additionally, you should submit your `MANUAL.pdf` file separately to **Markus**.

For P2, the total marks on Marmoset is 85. The remaining 15 marks is for the MANUAL (Section 7.3). Marmoset will continue to have separate public and release tests, as described in Section 6.4.

7.7 Evaluation

While developing your system, you should be aware that we shall be evaluating your system for the following:

1. Your code compiles on `linux.student.cs.uwaterloo.ca`.
2. Detailed documentation in the system manual (MANUAL.pdf).
3. We will use a number of test cases to test the following functionality in addition to Item 2 in Section 6.5:
   (a) Mutual exclusion on opens (write mode), and multiple opens by the same client to the same file that have not yet been closed
   (b) Concurrent readers and single writer to the same file
   (c) Accesses to the server version of the file are atomic
   (d) Client caches the file and respects the freshness condition
   (e) fsync triggers a flush of the file to the server

7.8 Tips

You should continue to follow the guidelines in Section 6.6.

Note that when a file is opened, it may exist at the server, but not at the client. The file should be created at the client which may necessitate additional changes to the flags passed to `open`.

Specific Do's and Don'ts

While doing **Project 2**, you can re-use your code from **Project 1**. **Do not** use code from a source other than your own **Project 1**.

Finally, we reiterate the importance of **starting this Project early**! There are many

small mistakes you can make that lead to unexpected and weird errors, and these take time to debug.

8 Frequently Asked Questions

8.1 What commands can I use to test my solution?

Section 6.3 shows examples of how to use bash commands (e.g. `echo`, `cat`, `touch`, `stat`) to test your WatDFS implementation. Alternatively, if you need more control over the calls you want to make, create test Python[10] or C++ programs that directly make the system calls you want. Doing so is a good way to replicate the tests yourself. You should look at the `watdfs.log` file to verify how your application calls are being translated into *libfuse* operations. You can figure out what system calls a tool like `cat` makes using `strace` (https://linux.die.net/man/1/strace).

8.2 How do I debug my code?

Consult the log files (`watdfs.log`, `rpc_client.log`, and `rpc_server.log`) to see what operations have taken place. Add debug print statements to your code, or use `gdb` to step through the operations that have taken place.

Important: When adding debug print statements, make sure to use the `DLOG` function and NOT `std::cout`.

Important: If you come to office hours, make sure you have tried to reproduce the error you're getting and have a sequence of logs that we can look at. Additionally, please ensure that you have submitted your (current) code to Marmoset so that the TAs can access it. **TAs cannot help you with programming issues without access to both the log trace and code submission on Marmoset.**

8.3 I tried debugging for hours but still cannot figure out why my program is failing. What should I do?

Keep the WatDFS architecture in mind. Recall Section 2, there are multiple calls being made—application->WatDFS client->server using RPC->file system—and back. The logs that we provide and your own print statements should be able to tell you which of these calls are failing. In particular:

- Did you test on `linux.student.cs.uwaterloo.ca`? Sometimes subtle errors discovered by Marmoset are hit only when executed in the same environment.
- Did the client and server initialize properly?
- Are all `rpcRegister` calls correct? Did the registrations succeed?
- Are the `rpcCall`s succeeding? Are you setting the correct RPC arguments and their types according to the protocol?
- Are you setting the correct arguments and flags to the function calls? In particular, check the arguments to the system calls (`open`, `stat`, etc) you're making. Did you set

[10] https://docs.python.org/3/library/os.html

the file path correctly?

- Are you returning the correct values? *libfuse* and in turn the applications depends on the return values from the `watdfs_client_*` functions for proper functioning. Unexpected return values are also a common source of bugs.
- Can you replicate the error (e.g., from the Marmoset results)? How is your test program behaving? Do you know that the expected behavior is? Can you spot in the logs which output is deviating from the expected behavior?
- Did you double check for common bugs: `null`s, forgetting to initialize variables, not allocating proper buffers space, use of wrong variables or incorrect order of use, and off-by-one errors?

8.4 Why can't I run `ls` or `mkdir`?

WatDFS supports only basic file operations, while `ls` and `mkdir` are directory operations. In general, you should check `watdfs.log`; if you see a call log containing `unimpl_*`, that means the operation is not supported by WatDFS. Note that we do not expect your WatDFS code to handle any directory operations.

8.5 Why aren't my RPC calls working?

Check your logs. Make sure that you are exporting the correct server address and port to your client, and that you are calling `rpcClientInit`. You can check what server address and port the client is trying to connect to by checking the `rpc_client.log` file. You can check what server address and port the server is listening on by checking the `rpc_server.log` file. More generally you should check the return codes from RPC library functions and compare them to the error codes defined in `rpc.h`. Finally, try comparing the argument types in the `rpc_client.log` and `rpc_server.log`, with the expected arguments registered at the server.

8.6 Why is FUSE trying to do file operations after I returned an error in `watdfs_cli_getattr`?

See Section 3.3. If you return an error in `watdfs_cli_getattr`, but there is information in the stat structure, then FUSE assumes the stat structure is correct. Make sure your stat structure has no data set in it if you are returning an error in these calls.

8.7 Why is FUSE returning the error: "fuse: bad mount point 'client_mount_dir': Transport endpoint is not connected"?

See Section 6.3. It is likely that there was a *libfuse* instance running which has not terminated (or it has crashed) that had `client_mount_dir` as a mount point. You should clean up state by running:

```
$ fusermount -u /path/to/client_mount_dir
```

8.8 Why does FUSE return "fuse: bad error value:" or "numerical result out of range"?

See Section 3.2. *libfuse* expects the return values from the functions to be 0 on success or <0 (`-errno`) if an error occurs. The exceptions are `watdfs_cli_read` and `watdfs_cli_write`. If your WatDFS client returns -1, `errno` will not be read by *libfuse*. As `errno` is positive, returning `errno` is also not acceptable.

8.9 Why do file operations return "Permission denied"?

There are two possible reasons. First, your WatDFS client may be returning `-1` instead of `-errno`. Note that `-EPERM` is `-1`, which indicates a permission error. Or second, new files created by your WatDFS code, either on the server or the client, might not have the right permissions by default, subsequently giving an error when trying to read those files. The `open` manpage has instructions on how to set file permissions explicitly.

8.10 Why do my Marmoset tests timeout?

It is likely that you are not specifying the correct `argTypes` in your `rpcCall`, or that the `args` do not match the `argTypes`. You should use the exact `argTypes` as defined in Section 4.1. You can look at the types that you are sending in the `rpc_client.log` file when you test locally.

8.11 How do I pick a directory to mount my client, store the server files, or cache data?

See Section 6.3. You should use directories that are not replicated (i.e., do not pick your home directory). You should also pick directories that are unique. The easiest directories to pick are subdirectories under `/tmp/$USER/`, which you may need to create.

8.12 My Marmoset test has an error, what do I do?

The output trace of the test will print out file-system operations that have succeeded and operations that have failed. You should devise your own test case that reproduces the failure locally, then debug your code.

If you are getting errors saying *invalid argument* or *numerical result out of range* for a system call such as `os.read()` or `os.open()`, they simply indicate that the WatDFS client/server did not return the expected response to the system call, resulting in Python throwing an exception.

See also 6.4.

8.13 Why isn't WatDFS working on my [local laptop, own Linux server, Mac]?

There could be a variety of reasons. E.g., your system could be missing `fuse.h` and other FUSE development files that needs to be installed separately.

Note that we can only provide support for running WatDFS on the CS Student Computing environment (linux.student.cs.uwaterloo.ca).

8.14 I am getting different/more calls than I expected?

FUSE has it's own internal logic to deal with some system calls. For example, a `mknod` call is preceded by a `getattr` call and if the file exists, the `mknod` call is simply dropped. Use Python to make granular system calls and keep checking `watdfs.log` to verify what exact calls your `watdfs_client` is getting from FUSE.