Distributed File System

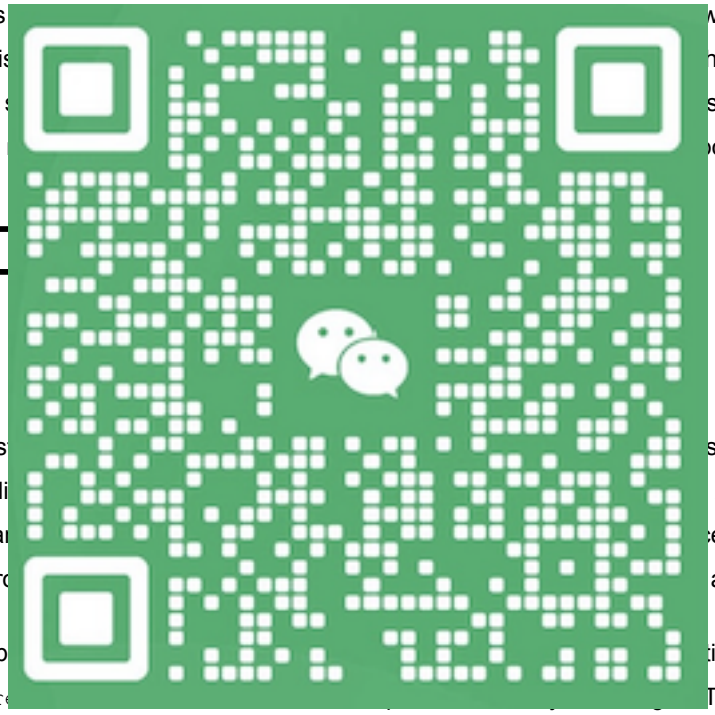
In this assignment, you will be developing a working distributed file server. We provide you with only the `gunrock_web` HTTP framework; you have to build the rest.

The goals of this project are:

- To learn the basics of on-disk structures for file systems
- To learn about file system internals
- To learn about distributed storage systems

This project consists of three main parts: reading on-disk storage using command line utilities, a local file system, and using the local file system to impelemnt a distributed file system. We recommend implementing your server in this order where you ensure that you have a solid foundation before moving on to the next part.

Fundamentally, this project is work with are all different. The disk operates only on di nical file system, and the API layer uses the local file s s have their own unique physical ove.



The main idea behind a dis same file system at the same time. One popular di Amazon's S3 storage system. S3 is used widely a essing data. With these basics in place, S3 pr all use every day.

Like local file systems, distrib ions. In this project you'll implement r TP APIs.

HTTP/REST API

In your distributed file system you will have two different entities: `files` and `directories`. You will access these entities using standard HTTP/REST network calls. All URL paths begin with `/ds3/`, which defines the root of your file system.

To create or update a file, use HTTP `PUT` method where the URL defines the file name and path and the body of your PUT request contains the entire contents of the file. If the file already exists, the PUT call overwrites the contents with the new data sent via the body.

In the system, directories are created implicitly. If a client PUTs a file located at `/ds3/a/b/c.txt` and directories `a` and `b` do not already exist, you will create them as a part of handling the request. If one of the directories on the path exists as a file already, like `/a/b`, then it is an error.

To read a file, you use the HTTP `GET` method, specifying the file location as the path of your URL and your server will return the contents of the file as the body in the response. To read a directory, you also use the HTTP `GET` method, but directories list the entries in a directory. To encode directory entries, you put each directory entry on a new line and regular files are listed directly, and directories are listed with a trailing "/". For `GET` on a directory omit the entries for `.` and `..`. For example, `GET` on `/ds3/a/b` will return:

```
c.txt
```

And `GET` on `/ds3/a/` will return:

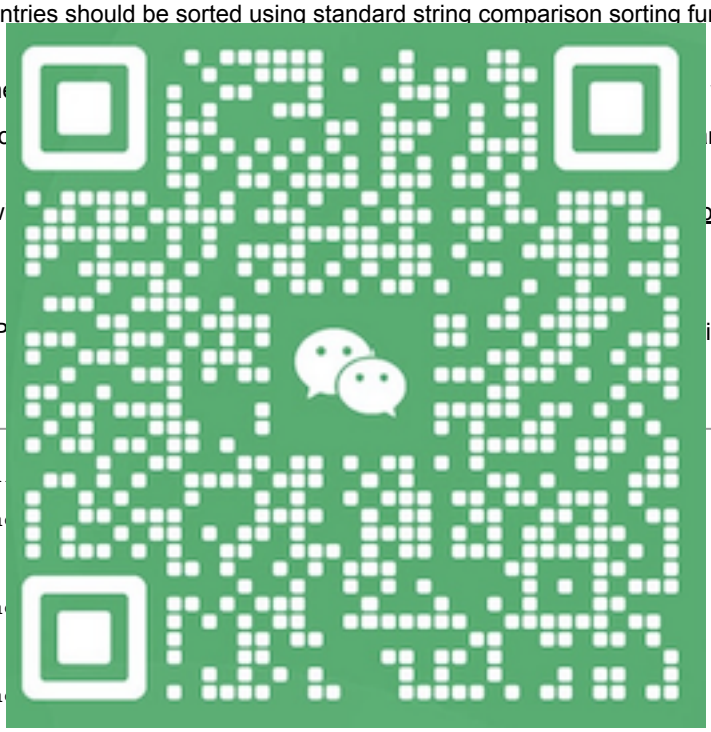
```
b/
```

The listed entries should be sorted using standard string comparison sorting functions.

To delete a file, you use the [illegible] your URL. To delete a directory, yo[illegible]n error.

You w[illegible]pp

Since Gunrock is a HTTP[illegible]it out. Here are a few



```
% curl -X PUT -d "fi
% curl http://localh
file contents
% curl http://localh
c.txt
% curl http://localh
c.txt
% curl http://localhost:8080/ds3/a
b/
% curl -X DELETE http://localhost:8080/ds3/a/b/c.txt
% curl http://localhost:8080/ds3/a/b/
%
```

Dealing with errors

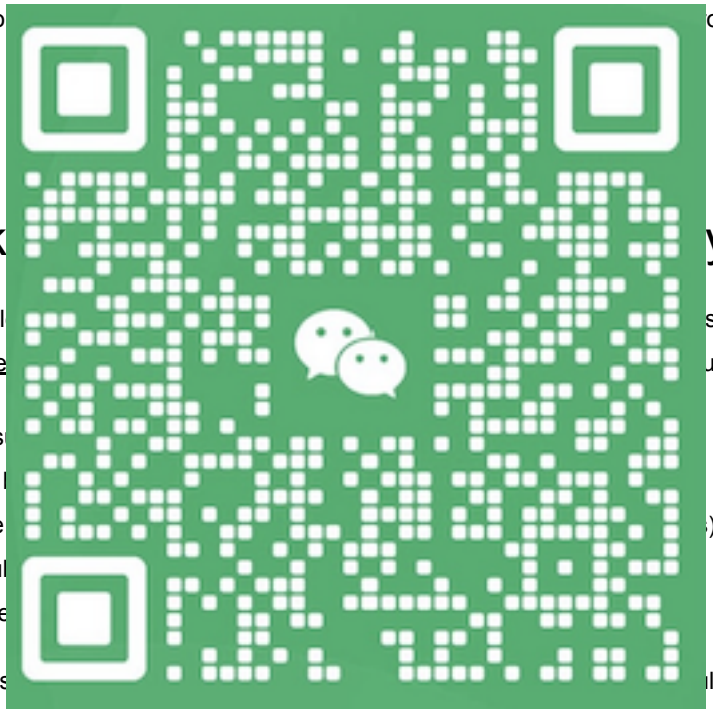
To implement your distributed storage interface, you will use a sequence of LocalFileSystem calls. Although each of these calls individually will ensure that they will not modify the disk when they have an error, since your implementation uses several LocalFileSystem calls it needs to clean up when something goes wrong. The key principle is that if an API

call has an error, there should not be any changes to the underlying disk or local file system.

To clean up the LocalFileSystem on errors, you can use the Disk (gunrock_web/include/Disk.h) interface for transactions. To use transactions, when you know that a file system call can change the disk's state, then start a transaction by calling `beginTransaction`. As your implementation for an API call proceeds, if the call is successful then you can `commit` your transaction to ensure that all file system modifications persist. If the call ends with an error, you can call `rollback` to reverse any writes that happened before the error.

There are four types of errors that your distributed file system can return. First, use `ClientError::notFound()` for any API calls that try to access a resouce that does not exist. Second, use `ClientError::insufficientStorage()` for operations that modify the file system and need to allocate new blocks but the disk does not have enough storage to satisfy them. Third, use `ClientError::conflict()` if an API call tries to create a directory in a location where a file already exists. Fourth, use `ClientError::badRequest()` for all other errors.

To return an error to the client, in your `DistributedFileSystemService.cpp` file, throw an exception using the ClientError (gunrock_web/include/ClientError.h) exception class, and the gunrock web framework will catch these errors and co or.



On-Disk ystem

The on-disk fil ssed here (https://pages.cs.wisc.e ures are as follows:

- A single block (4KB) s
- An inode bitmap (can
- A data bitmap (can be)
- The inode table (a mul
- The data region (some

More details about on-disk s ld use. Specifically, this has a very specific format for the super block, inode, and directory entries. Bitmaps just have one bit per allocated unit as described in the book.

As for directories, here is a little more detail. Each directory has an inode, and points to one or more data blocks that contain directory entries. Each directory entry should be simple, and consist of 32 bytes: a name and an inode number pair. The name should be a fixed-length field of size 28 bytes; the inode number is just an integer (4 bytes). When a directory is created, it should contain two entries: the name `.` (dot), which refers to this new directory's inode number, and `..` (dot-dot), which refers to the parent directory's inode number. For the root directory in a file system, both `.` and `..` refer to the root directory.

When your server is started, it is passed the name of the file system image file. The image is created by a tool we provide, called `mkfs`. It is pretty self-explanatory and can be found here (gunrock_web/mkfs.c).

When accessing the files on an image, your server should read in the superblock, bitmaps, and inode table from disk as needed. When writing to the image, you should update these on-disk structures accordingly.

One important aspect of your on-disk structure is that you need to assume that your server can crash at any time, so all disk writes need to leave your file system in a consistent state. To maintain consistency you'll need to order your writes carefully to make sure that if your system crashes your file system is always correct.

Importantly, you cannot change the file-system on-disk format.

For more detailed documentation on the local file system specification, please see LocalFileSystem.h (gunrock_web/include/LocalFileSystem.h) and the stub LocalFileSystem.cpp (gunrock_web/LocalFileSystem.cpp). Also, please see Disk.h (gunrock_web/include/Disk.h) for the interface for accessing the disk.

Bitmaps for block allocation

We use on-disk bitmaps to keep track of entries (inodes and data blocks) that the file system has allocated. For our bitmaps for each byte, the least significant bit (LSB) is considered the first bit, and the most significant bit (MSB) is considered the [redacted] this in hex:

```
byte position:  0  1
hex value:     01 00

bit position   0
bit value:     1 0 0
```

```
byte position:  0  1
hex value:     00 80

bit position   0
bit value:     0 0 0
```

LocalFileSystem `write` and `read` semantics

In our file system, we don't have a notion of appending or modifying data. Conceptually, when we get a `write` call we overwrite the entire contents of the file with the new contents of the file, and write calls specify the complete contents of the file.

Calls to `read` always read data starting from the beginning of the file, but if the caller specifies a size of less than the size of the object then you should return only these bytes. If the caller specifies a size of larger than the size of the object, then you only return the bytes in the object.

LocalFileSystem out of storage errors

One important class of errors that your `LocalFileSystem` needs to handle is out of storage errors. Out of storage errors can happen when one of the file system modification calls -- `create`, `write`, and `unlink` -- does not have enough availabe storage to complete the request. You should identify out of storage errors before making any writes to disk. So in other words, your file system should be unchanged if an out of storage error happens.

File system utilities

To help debug your disk images, you will create three small command-line utilities that read information about a given disk image and write it out to the command line. We have included an example disk image and expected outputs in the disk_testing (gunrock_web/disk_testing) directory, but make sure that your utilities can handle multiple different disk image configurations and contents.

To implement your file system utilities, you'll want to have implementaitons for read-only functions within `LocalFileSystem.cpp` a should implement `stat` and `read` to implement `ds` in our file system). For `ds3bits` you'll war `dDataBitmap`.

Our hope is that with these u can use these utilities to

Note: We will only test your disk in the test cases is

The `ds3ls` utility prints the n takes a single command line argument: the name c e file system, print the contents of that directory in fu ss repeats in a depth-first

When printing the contents of a directory, first, print the word `Directory` followed by a space and then the full path for the directory you're printing, and ending it with a newline. Second, print each of the entries in that directory. Sorted each entry using `std::strcmp` and print them in that order. Each entry will include the inode number, a tab, the name of the entry, and finishing it off with a newline. Third, print a empty line consisting of only a newline to finish printing the contents of the directory.

Make sure that your solution does *not* print the contents of `.` and `..` subdirectories as this action would lead to an infinitely loop.

After printing a directory, traverse into each subdirectory and repeat the process recursively in a depth-first fashion.

The `ds3cat` utility

The `ds3cat` utility prints the contents of a file to standard output. It takes the name of the disk image file and an inode number as the only arguments. It prints the contents of the file that is specified by the inode number.

For this utility, first print the string `File blocks` with a newline at the end and then print each of the disk block numbers for the file to standard out, one disk block number per line. After printing the file blocks, print an empty line with only a newline.

Next, print the string `File data` with a newline at the end and then print the full contents of the file to standard out. You do not need to differentiate between files and directories, for this utility everything is considered to be data and should be printed to standard out.

The `ds3bits` utility

The `ds3bits` utility prints metadata for the file system on a disk image. It takes a single command line argument: the name of a disk image file.

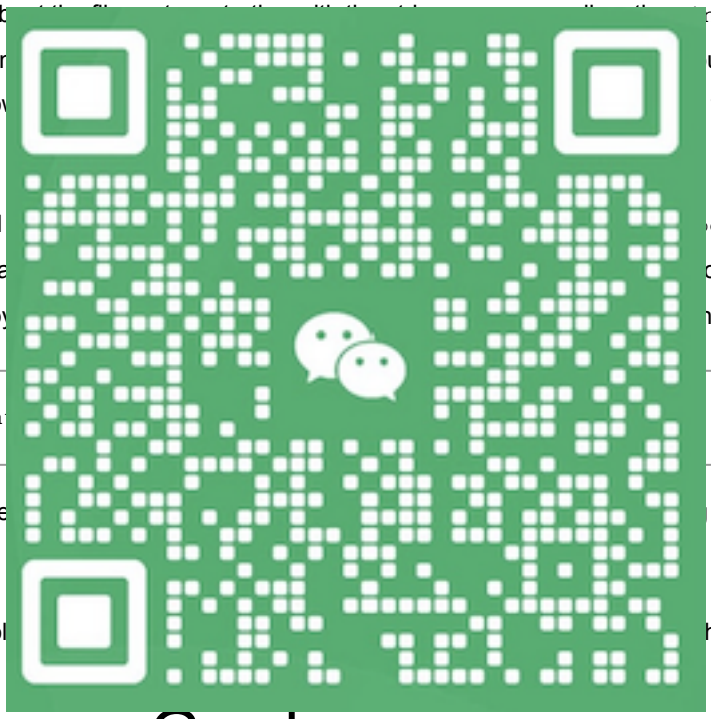
First, it prints metadata ab `node_region_addr` followed by a space ar uld print the string `data_region_addr` follo followed by a newline.

Next it prints the inode and `de bitmap` and `Data bitmap` respectively. For ea ollowed by a space. For each b is:

```
cout << (unsigned in
```

Where each byte is followe all of the bytes print a

Print the indoe bitmap first, fol hen print the data bitmap.



Gradescope

We are using Gradescope to autograde your projects. You should submit the following files to Gradescope: `LocalFileSystem.cpp`, `DistributedFileSystemService.cpp`, `ds3ls.cpp`, `ds3cat.cpp`, and `ds3bits.cpp`. The autograder requires all files to run so we provided stubs for each of these that you can use until you have your own implementation of them.