CS 4513 - Distributed Systems
Project 1 (115 points)

Project: Goat File System

In this project, you will get experience with file system management by implementing the Goat file system. From the perspective of the calling application, the Goat file system behaves much like the built-in Linux file system and implements the following functions:

1. `debug()`: This function scans a mounted filesystem and reports on how the inodes and blocks are organized.
2. `format()`: This function creates a new filesystem on the disk, destroying any data already present. Specifically, it will initialize the superblock, the inode table, and set aside ten percent of the blocks for inodes. It must return `true` on success, `false` otherwise.
3. `mount()`: This function examines the disk for a filesystem. If one is present, it will read the superblock, build a free block bitmap, and prepare the filesystem for use. It must return `true` on success, `false` otherwise.
4. `create()`: This function creates a new inode of zero length. It must return `inumber` on success, `-1` otherwise.
5. `wremove(size_t inumber)`: This function removes the inode indicated by the `inumber`. Specifically, it should release all data and indirect blocks assigned to this inode and return them to the free block map. It must return `true` on success, `false` otherwise.
6. `stat(size_t inumber)`: This function returns the logical size of the given `inumber`, in bytes. Note that zero is a valid logical size for an inode. On failure, it returns -1.
7. `wfsread(size_t inumber, char *data, size_t length, size_t offset)`: This function reads data from a **valid inode** indicated by the `inumber`. It then copies `length` bytes from the data blocks of the inode into the `data` pointer, starting at `offset` in the inode. It should return the total number of bytes read. If the given `inumber` is invalid, or any other error is encountered, the method returns -1.
8. `wfswrite(size_t inumber, char *data, size_t length, size_t offset)`: This function writes data to a **valid inode** indicated by the `inumber` by copying `length` bytes from the `data` pointer into the data blocks of the inode, starting at `offset` in the inode. It will allocate any necessary direct and indirect blocks in the process. It should return the total number of bytes written to the disk. If the given `inumber` is invalid, or any other error is encountered, the method returns -1.

These functions are prototyped in the provided `goatfs.h` header file.

There are a number of additional files that are provided to facilitate the development and tests:

1. `disk.h` and `disk.c` emulate a disk by only allowing the Goat file system to read and write in blocks of predefined size. For this project, we emulate the disk by dividing a normal file (referred to as *disk image*) into 4096-byte blocks.
2. `test_goatfs.c` is a wrapper that facilitates the testing process. It can be used directly in command line or be invoked by the provided bash test scripts (see below). In essence, it parses and maps the command line arguments to corresponding file system operations.
3. `data/image.5`, `data/image.20`, and `data/image.200` are pre-formatted disk images with 5, 20, and 200 blocks respectively. All test cases will be based on these three disk images.

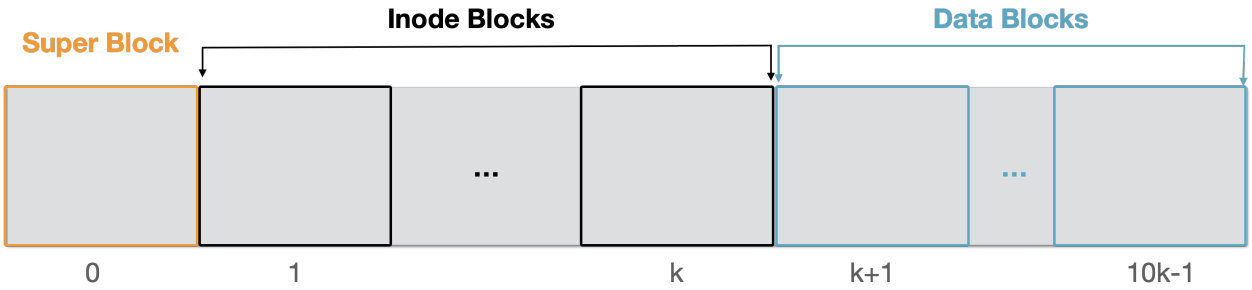

Figure 1: Overview of disk image managed by the Goat file system. In this example, the disk image consists of 10K blocks. Each block is 4096 bytes.

4. `tests/test_*.sh` are a collection of bash scripts that we will be using to test file system functionalities.

There are three critical restrictions for your implementation. Any projects failing to adhere to these restrictions are likely to receive a score of 0 points.

- You **must not delete** any of the provided code, and your file system code should be put in the `goatfs.c` file. However, it is quite likely that you will find additional variables or functions useful in implementing this simple file system. Feel free to `add` them to the `goatfs.h` header file.
- Your Goat file system **must** call the provided disk emulator, such as read (using `wread()`) and write (using `wwrite()`) to the disk image in 4KB blocks, to implement the required file system functions.
- The super block and the inode blocks must be stored and managed by your Goat file system.

Goat File System Design Overview

The following descriptions are encoded in the header files ***`disk.h`*** *and* ***`goatfs.h`****, respectively.*

The Goat file system uses the disk layout illustrated in Figure 1. The disk is divided to 4KB blocks. In this example, we have 10K blocks.

The first block, called super block, is reserved for file system meta data that describes the layout of the rest file system. Specifically, a total of 16 bytes are used to keep track of four pieces of information in the following order: *(i)* **the magic number** which can be used to check whether the filesystem is valid or not. This check is often done when the OS tries to mount the file system. *(ii)* **the number of blocks** which represents the total number of blocks. *(iii)* **the number of inode blocks**, denoted as N, which describes the number of blocks used for storing inodes. In this project, we will allocate 10% of the total number of blocks (rounding up) to serve as inode blocks. *(iv)* **the total number of inodes** which describes the number of inodes across all the inode blocks.

Note that most of block 0 is left unused as the super block data structure only takes 16 bytes.

For this project, we will use 32 bytes to store the inode data structure. This means that for each inode block, we can store 128 inodes. Figure 2 shows an example of the eight fields that an inode keeps track of. Specifically, *(i)* **valid** describes whether an inode has been created (i.e., valid) or not. If an inode is valid, this field has the value of 1. *(ii)* **size** denotes the logical size of the inode data in bytes (i.e., the size of file represented by each inode). *(iii)* **direct** which is an integer array with five elements. Each element stores the block number where data maybe found. For example, if `direct[0] = 2`, this means that the first 4KB data is stored in data block 2. Additionally, we will use 0 to denote that the element does not point to any data block. For example, if `direct[1] = 0`, this means that the current inode only points to one data block; this also indicates that the logical size of the inode is at most 4KB. *(iv)* **indirect** which stores the number of block used for storing additional block numbers. As we talked about in class, having the indirect

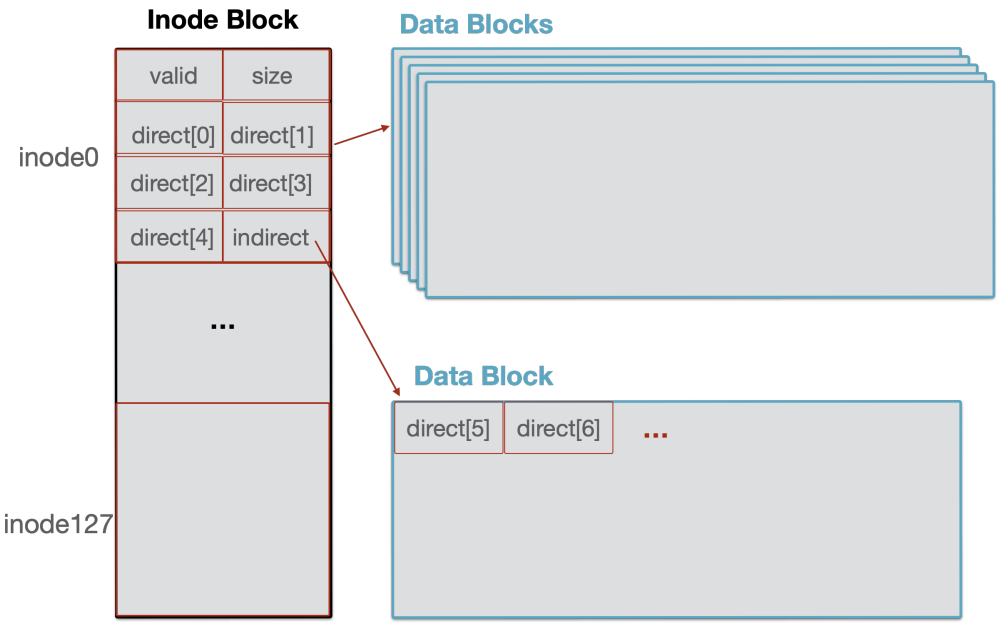

Figure 2: Overview of disk image managed by the Goat file system. In this example, the disk image consists of 10K blocks. Each block is 4096 bytes.

field allows storing bigger files, e.g., beyond 20KB. Note that the indirect field is not set unless all elements of `direct[]` are utilized.

Different from a real-world file system, our Goat file system will keep an in-memory free block bitmap which keeps track whether a data block is available or in use. This bitmap would be consulted and updated every time the Goat file system needs to add or remove a data block from an inode. Because we use an in-memory data structure for the bitmap, each time that a Goat file system is mounted, it must build a new free block bitmap from scratch by scanning through all of the inodes and recording which blocks are in use.

Hints

Some concepts that might be useful to review before starting this project:

- Review OSTEP Chapter 39 and Chapter 40 before starting this project. It introduces many of the concepts that you will need to understand to implement the Goat file system. For example, Chapter 40 talks about at length about overall file system structure and inode information.
- While OSTEP Chapter 39 and Chapter 40 are useful, the Goat file system works a little differently than the schemes described in the book by making a number of simplifying assumptions. For example, the Goat file system will not support directory. Take note of the differences and think about how that will change your implementation.
- We recommend you follow test-driven development. In essence, before attempting any implementations, first read and understand the corresponding test cases.
- We recommend you implement the required functions roughly in order as they are described in the project specification. For example, first attempts the easier functions `debug()`, `format()`, and `mount()` to make sure that you can successfully interact with the provided disk images, before moving on to the remaining functions.

- During the development phase, we recommend using the wrapper `./test_goatfs` to perform test cases **individually**. You can extract the information about how to invoke the test cases by reading the `test_goatfs.c` and the expected output in the `tests/test_*.sh`. To avoid unnecessary confusions, we recommend always perform individual tests on a backup disk image, e.g., `image.5.tmp`.

Debug A Disk Image

The following requirements are encoded in test cases `test_debug.sh`.

The `debug()` function must make a call to the provided disk emulator (see `disk.h`) to scan the disk image and output how the inodes and blocks are organized. For simplicity, the disk image is defined as `DISK* _disk` in the `goatfs.h`.

Calls to `debug()` should produce output similar to the following:

```
$ ./test_goatfs ../data/image.5 5 debug
SuperBlock:
    magic number is valid
    5 blocks
    1 inode blocks
    128 inodes
Inode 1:
    size: 965 bytes
    direct blocks: 2
2 disk block reads
0 disk block writes
```

In the above example, the Goat file system issues two read operations to the disk emulator. The first read retrieves information relevant to the SuperBlock. Once correctly parsed, it indicates that this disk image has 5 blocks and uses 1 block for inodes. Therefore, to display the inode information, a second read is issued. By parsing the retrieved block, the Goat file system displays the information about Inode 1—which is the only Inode with the valid field sets to 1.

Again, your file system cannot bypass the provided disk emulator to directly interact with the disk image.

Format A Disk Image

The following requirements are encoded in test cases `test_format.sh`.

The `format()` function must call the provided disk emulator to create a new filesystem on a disk image.

The disk emulator will provide you with essential information such as the number of blocks that you will need for formatting the Goat file system. Recall that we will reserve 10% of the blocks for inodes (rounding up). Note that formatting a filesystem does not cause it to be mounted. Further, an attempt to format an already-mounted disk should do nothing and return failure.

Calls to `format()` should produce output similar to the following:

```
$ cp ../data/image.5 ../data/image.5.tmp
$ ./test_goatfs ../data/image.5.tmp 5 format
disk formatted.
SuperBlock:
    magic number is valid
    5 blocks
    1 inode blocks
    128 inodes
2 disk block reads
5 disk block writes
```

In the above example, we first create a copy of the provided disk image with the `cp` command. Then, we perform a disk format operation on the `image.5.tmp` followed by a debug operation. In other words, the output file system information reflects the state of the newly formatted file system.

Mount A Disk Image

The following requirements are encoded in test cases ***test_mount.sh****.*

The `mount()` function must call the provided disk emulator to scan for the Goat file system. It will proceed by reading the first block and trying to interpret the bytes based on the super block semantic described in the Overview and Figure 1.

Consequently, it will initialize an in-memory free block bitmap by reading in inode blocks. As such, a successful mount is a pre-requisite for the remaining functions described below—the requirements are reflected in the test cases.

Calls to `mount()` should produce output similar to the following:

```
$ ./test_goatfs ../data/image.5 5 mount
disk mounted.
2 disk block reads
0 disk block writes
```

Create An inode

The following requirements are encoded in test cases ***test_create.sh****.*

The `create()` function will walk through the inode table to find the next available inode. It will then perform the necessary inode initialization such as setting the direct pointers and size. Finally, it will save the updated inode structure back to the disk image.

Calls to `create()` should produce output similar to the following:

```
$ cp ../data/image.5 ../data/image.5.create
$ ./test_goatfs ../data/image.5.create 5 create
SuperBlock:
  magic number is valid
  5 blocks
  1 inode blocks
  128 inodes
Inode 1:
  size: 965 bytes
  direct blocks: 2
disk mounted.
created inode 0.
created inode 2.
//[output is truncated...]
create failed!
SuperBlock:
  magic number is valid
  5 blocks
  1 inode blocks
  128 inodes
Inode 0:
  size: 0 bytes
  direct blocks:
Inode 1:
  size: 965 bytes
  direct blocks: 2
//[output is truncated...]
Inode 126:
  size: 0 bytes
  direct blocks:
Inode 127:
  size: 0 bytes
  direct blocks:
261 disk block reads
127 disk block writes
```

In the above example, note we first create a copy of the provided disk image with the `cp` command. Also note that we do not create the *inode 1* because it is not available! **Hint:** what happens if the inode table is full? The creation will fail!

Remove An inode

The following requirements are encoded in test cases ***test_remove.sh****.*

The `remove(size_t inumber)` function is the counterpart to the `create()` function—it will remove the the inode indicated by the `inumber`. This function should reset all the blocks used by the inode and update the free block bitmap.

Calls to `remove(size_t inumber)` should produce output similar to the following:

```
$ cp ../data/image.5 ../data/image.5.remove
$ ./test_goatfs ../data/image.5.remove 5 remove-test-1
//[output is truncated...]
disk mounted.
created inode 0.
created inode 2.
created inode 3.
removed inode 0.
//[output is truncated...]
created inode 0.
removed inode 0.
remove failed!
removed inode 1.
removed inode 3.
SuperBlock:
  magic number is valid
  5 blocks
  1 inode blocks
  128 inodes
Inode 2:
  size: 0 bytes
  direct blocks:
25 disk block reads
8 disk block writes
```

In the above example, we leverage the `create()` function to create a number of inodes and then subsequently calling `remove(size_t inumber)` by passing in valid/invalid inumbers.

Stat An inode

The following requirements are encoded in test cases ***test_stat.sh****.*

The `stat(size_t inumber)` function is quite simple—it simply read in the inode information indicated by the `inumber` via the disk emulator. It then checks to see if the inode is valid or not before returning the logical size of an inode. Note that zero is a valid logical size for an inode.

Calls to `stat(size_t inumber)` should produce output similar to the following:

```
$ ./test_goatfs ../data/image.5 5 stat-test-image.5
disk mounted.
inode 1 has size 965 bytes.
stat failed!
stat failed!
5 disk block reads
0 disk block writes
```

In the above example, it shows that calling `stat(size_t inumber)` on valid inode (i.e., inode 1) reports its logical size while calling on invalid inodes will fail!

Read from An inode

The following requirements are encoded in test cases ***test_copyout.sh****.*

The `wfsread(size_t inumber, char *data, size_t length, size_t offset)` function reads data from a **valid inode** indicated by the `inumber`. The number of requested bytes is specified by the `length`. Depending on the file size, the number of bytes actually read (from the emulated disk) could be smaller than the number of bytes requested, perhaps if the end of the inode is reached. Conceptually, this function provides user-level applications the means to retrieve (previously stored) files from disk.

Calls to `wfsread(size_t inumber, char *data, size_t length, size_t offset)` should produce output similar to the following:

```
$ ./test_goatfs ../data/image.5 5 copyout-test-1 /tmp
disk mounted.
965 bytes copied
5 disk block reads
0 disk block writes
$ md5sum /tmp/1.txt | awk '{print $1}'
1edec6bc701059c45053cf79e7e16588
```

In the above example, we are copying out an inode to a file `/tmp/1.txt` by calling the `wfsread` function until the end of the inode is reached. We then use the `md5sum` utility to output the file's checksum which will further be used to verify the correctness of implementation.

Write to An inode

The following requirements are encoded in test cases ***test_copyin.sh****.*

The `wfswrite(size_t inumber, char *data, size_t length, size_t offset)` function is the counterpart to the previous `wfsread` function. Note, the number of bytes actually written (to the emulated disk) could be smaller than the number of bytes request, perhaps if the disk becomes full. Conceptually, this function provides user-level applications the means to save files to disk.

Calls to `wfswrite(size_t inumber, char *data, size_t length, size_t offset)` should produce output similar to the following:

```
$ cp ../data/image.5 ../data/image.5.copyin
$ ./test_goatfs ../data/image.5.copyin 5 copyin-test-1 /tmp
SuperBlock:
  magic number is valid
  5 blocks
  1 inode blocks
  128 inodes
//[output is truncated...]
965 bytes copied
19 disk block reads
4 disk block writes
$ md5sum /tmp/1.copy | awk '{print $1}'
1edec6bc701059c45053cf79e7e16588
```

In the above example, we are verifying the correctness of the `wfswrite` implementation by copying in a file `1.txt` to the disk followed by a copying out to `/tmp/1.copy`. If this function is implemented correctly, the checksums produced by `md5sum` should stay the same.

Test Cases

We have provided three disk images in the `data` directory to help you verify your Goat file system implementation. **All the test cases require the unmodified disk images. Therefore, you should make sure to these images to their original states before running the provided test cases.**

We have provided a suite of test cases that define and exercise the critical functionality of your Goat file system. These test cases are all defined in the `test_*.sh` scripts under the `tests` directory, which in turn calls the wrapper `test_goatfs`.

To simplify development, we recommend that you use the test cases to guide your development and implement the file system functions roughly in order (as defined in `goatfs.h`). For example, first implement the `debug` function to pass the test cases (`test_debug.sh`), and then moving on to implement the `format` function to pass the test cases (`test_format.sh`).

You can invoke the test scripts as following:

```
$ ./tests/test_debug.sh
Testing debug on data/image.5 ...  passed
Testing debug on data/image.20 ...  passed
Testing debug on data/image.200 ...  passed
```

In cases where a test fails, you would likely need to consult both the `test_*.sh` test script and the wrapper code `test_goatfs.c` for information on inputs and expected outputs.

Your file system must pass provided test cases, which you can run by doing `make test`, to receive full credit for the assignment. The provided test cases are by no means comprehensive and you are encouraged to write your own test cases. You can add your new test cases following similar structures by appending the `test_goatfs.c` and adding new bash test scripts.

Finally, the behavior implied by these test cases supersedes any perceived contradictions or ambiguity in the written specifications. In other words, if anything is unclear in the specifications, please refer to the test cases for the intended behavior.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any checkpoint assignment which passes the debug, format, mount, and create test cases will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting the project, please include the following:

- All files that were provided in the starter code—**please do not change the directory layout as the both the Makefile and bash scripts rely on this layout information**.
- The source code of your Goat File System in a file named `goatfs.c`.
- The `README.txt` which describes your design choices. Only plaintext write-ups are accepted; Markdown is allowed.
- (Optional:) Any additional test cases you developed. Please include them in the `tests/` directory and add the corresponding command line parsing to the `test_goatfs.c`.

Please compress all the files together as a single .zip, named `YOUR_WPI_USERNAME_project_fs.zip`, archive for submission. As with all projects, please **only use standard zip files** for compression; **.rar, .7z, and other custom file formats will not be accepted**.

Grading

A grading rubric has been provided at following table to give you a guide for how the project will be graded.

Task ID	Description	Points
1	Passed all tests in `test_debug.sh`; 6 point(s) per test	18
2	Passed all tests in `test_format.sh`; 3 point(s) per test	9
3	Passed all tests in `test_mount.sh`; 2 point(s) per test	14
4	Passed all tests in `test_create.sh`; 6 point(s) per test	6
5	Passed all tests in `test_remove.sh`; 6 point(s) per test	6
6	Passed all tests in `test_stat.sh`; 2 point(s) per test	6
7	Passed all tests in `test_copyout.sh`; 6 point(s) per test	18
8	Passed all tests in `test_copyin.sh`; 6 point(s) per test	30
9	The design document (as part of README.txt); 1 point(s) per design	8

Note: The final project grade will also be adjusted based on the checkpoint completion as well as the late submission. passing all tests are not sufficient to earn all the points. The correctness of the implementation is judged at the discretion of the grader. As an example, any assignment that leverages hardcode to produce the expected output will lead to point deductions.