CS 423 Spring 2026 MP4: File System Checking

Assignment Due: May 5th at 11:59 PM CT. No late submission. No regrade.

Overview

In this assignment, you will be developing a working file system checker. A checker reads in a file system image and makes sure that it is consistent. When it isn't, the checker takes steps to repair the problems it sees. The basic version of your MP won't need to make any repairs to the file system image. However, if you're up for a challenge and would like to earn some extra credits, you're welcome to try adding that feature.

Before you start

What you need

- Linux kernel will not be used in this MP, however, you should still code and test this MP on EngIT VMs. **Our auto-grader will test your code on the EngIT VM.**
- You should be able to read, write, and debug program code written in C language.

Background

Some basic background about file system consistency is found in the textbook chapter on Crash Consistency: FSCK and Journaling

Problem Description

The myfsck checker

For this project, you will read and check file system images. The file system format is somewhat similar to VSFS (discussed in the Apr14 lecture). The file `include/fs.h` contains the data structures you need to understand, including the superblock, on disk inode (`struct dinode`), and directory entry (`struct dirent`). You should also read through the tool `mkfs.c` to see

how an empty file-system image is generated. You can use that same tool to generate a file system image with a dataset by invoking "`mkfs fs.img <your-dir-tree>`". The tool lsfs.c can be used to walk the file system; invoking "`lsfs fs.img`" will recursively list all the files in that image.

Much of this project requires figuring out the exact on-disk format used by this file system, and then coding up checks to see if various data structures are consistent. Thus, reading through `mkfs.c` and the file system code will help you understand how the various bits and bytes in the image are used to persist the file system.

Your checker should read through the file system image and determine the consistency of a number of things, including the following. When a problem is detected, report it using **ONLY the error functions provided in the** `log.h` file, and exit immediately with exit code 1 (i.e., call `exit(1)`).

IMPORTANT NOTE: The auto-grader expects the exact errors provided in `log.h`. So, please use ONLY those functions to output the errors listed in the cases below. For example, in #1 below, invoke `superblock_error()`.

1. For the metadata in the super block, the file-system size is larger than the number of blocks used by the super-block, inodes, bitmaps and data. If not, print `MP4_ERR: superblock is corrupted.`

2. Each inode is either unallocated or one of the valid types (`T_FILE`, `T_DIR`, `T_DEV`). If not, print `MP4_ERR: bad inode.`

3. For in-use inodes, each address that is used by inode is valid (points to a valid data block address within the image). If the direct block is used and is invalid, print `MP4_ERR: bad direct address in inode.`; if the indirect block is in use and is invalid, print `MP4_ERR: bad indirect address in inode.` If any of the addresses within the indirect block is in use and is invalid, print `MP4_ERR: bad indirect address in inode.`

4. Each directory contains `.` and `..` entries, the `.` entry points to the directory itself. (NOTE: The root directory `/` should also have both entries). If not, print `MP4_ERR: directory not properly formatted.`

5. For an in-use inode, each address pointed to by the inode is also marked as used in the bitmap. If not, print `MP4_ERR: address used by inode but marked free in bitmap.`

6. For a block that is marked as in-use in the bitmap, it should actually be in-use in some inode or indirect block. If not, print `MP4_ERR: bitmap marks block in use but it is not in use.`

7. Each direct address must be used only once across all in-use inodes. If not, print `MP4_ERR: direct address used more than once.`

8. For each in-use inode, the file size must be within the actual number of blocks used for storage. That is, if `b` blocks are used with block size `s`, then the file size must be `> (b-1)*s` and `<= b*s`. If not, print `MP4_ERR: incorrect file size in inode.`

9. For each inode marked in use, it must be referred to in at least one directory. If not, print `MP4_ERR: inode marked used but not found in a directory.`

10. For each inode number that is referred to in a valid directory, it must be marked in use. If not, print `MP4_ERR: inode referred to in directory but marked free.`

11. Reference count (number of links) to a regular file must match the number of times a file is referred to in all directories (i.e., hard links to that file is correct). If not, print `MP4_ERR: bad reference count for file.` (NOTE: this file system does not support for soft links)

12. No extra links are allowed to directories, i.e., each directory must appear exactly in one other directory. If not, print `MP4_ERR: directory appears more than once in the file system.`

Other Specifications

Your checker program, called `myfsck`, must be invoked exactly as follows:

```
None
prompt> myfsck file_system_image
```

The image file is a binary file that contains the file system image. If no image file is provided, you should print the usage error shown below:

```
None
prompt> myfsck
Usage: myfsck <file_system_image>
```

This output must be printed to standard error and exit with the error code of 1.

If the file system image does not exist, you should print the error `image not found.` to standard error and exit with the error code of 1.

If the checker detects any one of the 12 errors above, it should print the specific error to standard error and exit with error code 1.

If the checker detects none of the problems listed above, it should exit with a return code of 0 and not print anything.

We have provided one consistent file system image called `good.img` and one corrupted file system image called `bad_direct_addr.img` (for point #3 from the previous section). You are encouraged to hack/change the `mkfs` tool to generate inconsistent file system images that you can use to test & validate your `myfsck`.

Extra Credits

For this project, you can gain extra points (**10% of the MP**) by implementing the following more challenging condition checks:

1. Each `..` entry in a directory refers to the proper parent inode and parent inode points back to it. `MP4_ERR: parent directory mismatch`.
2. Every directory traces back to the root directory, i.e., there must be no loops in the directory tree. `MP4_ERR: inaccessible directory exists`.
3. Repair the `"inode marked used but not found in a directory"` error (Point #9 in the prior section). Your MP must collect these files and/or directories and put them in the `lost_found` directory. You must create this directory under the root directory of the provided image. To create this directory, you can use the `add_dir()` function in `mkfs.c` or create another version of it, as you see fit.
4. The structure of the `lost_found` should be `/lost_found/#<directory_inum>/<files…>`, because it is impossible to recover the directory name.
5. Repair the incorrect reference counts (Point #11 in the prior section) for any inode by setting it to the correct number of links.

For these last two repairs you will need to obtain write-access to file system image in order to modify it. The repair operation of the program should be performed only when the `-r` flag is specified.

```
Shell
prompt> myfsck -r image_to_repair
```

Hints

For this MP, we **strongly recommend** that you use `mmap()` for reading/writing from/to the file system image. It will make your life much easier!

Make sure to look at `fs.img`, which is a consistent file system image created by the `mkfs` tool. You can use the `lsfs` tool for this purpose. The tests, of course, will create inconsistencies in these images, but your tool should work over a consistent image as well. Study `mkfs` and its output to start making progress on this project.

Compile and Test Your Code

A Makefile has been provided. Use "`make lsfs`" & "`make mkfs`" to create the two aforementioned tools. Use "`make myfsck`" to compile your code.

It is a good habit to get basic functionalities working before moving to advanced features. As mentioned earlier, we have provided only 2 sample file system images: a consistent `good.img` and an inconsistent `bad_direct_addr.img` that you can use for your early tests. You are expected to generate inconsistent file system images for testing the other cases.

The test file system images for the extra credit part will not be provided. Like in the earlier test cases, you are expected to create your own images and scripts to test your implementation.

Submit Your Result

Here are the steps to accept and submit your MP.

- Open the link https://classroom.github.com/a/My8SEkXV and log in using your GitHub account.

- Find your name in the student list and click it to accept the assignment. Please double-check your name and email address before accepting the assignment (If you choose another's name by mistake, please contact TA).
    - If you can't find your name in the list, please contact TA
- A repo named `cs423-uiuc/mp4-<GitHubID>` will be automatically created for you with the starter code in it.
- You are free to create any extra header files. As long as the code compiles with `gcc myfsck.c`. In other words, no other `c` files should be needed to create the binary.
- For `types.h` and `fs.h,` you should use the one provided without any changes.

Grading Criteria

Criterion	**Points**
Your code compiles with no warning	10
Readme	5
Your code is well commented, readable, and follows software engineering principles.	10
Catch all errors	75
Catch all errors (extra credit)	6
Repair all errors (extra credit)	4
Total	**100 + 10 (EC)**