Middle East Technical University | Department of Computer Engineering

CENG 334

Operating Systems

Spring 2023-2024 - Homework 3

File System Integrity in an EXT2 file system

Due date: 30 May 2024, Thursday, 23:59

1 Introduction

In this assignment, you are tasked with implementing a corruption recovery program for the ext2 file system. In this endeavor, you will learn about ins-and-outs of ext2, including how to traverse its directory hierarchy, how to locate the contents of files and how to interpolate missing information using the capabilities of the ext2 file system.

2 ext2

2.1 File System Details

The ext2 file system was introduced for Linux in 1993 and was designed based on Unix file system (UFS) principles, which are covered in your course. Understanding the details of ext2 is crucial for comprehending how the extension will function and for writing your code. Although newer versions of Linux use the ext4 file system, ext3 and ext4 are mainly extensions that add journaling and modern features. The fundamental concepts and data structures of ext2 remain the same.

Structure definitions are provided for your use in the `ext2fs.h` header file and some are also shown below.

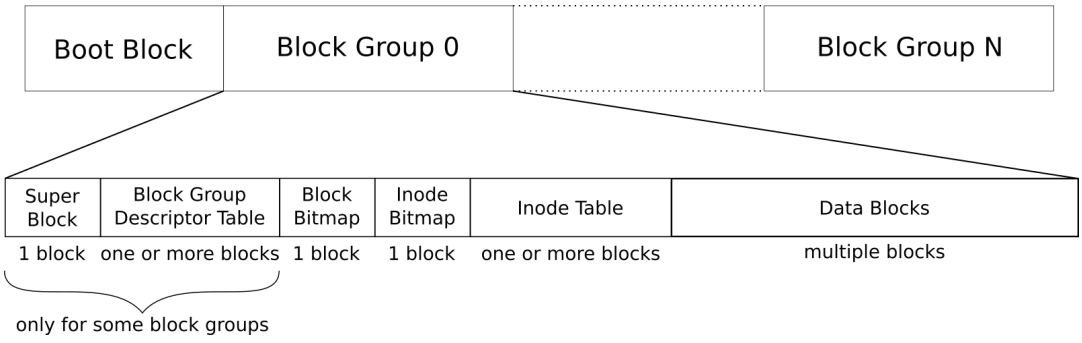

Figure 1: Overall ext2 Layout

An *example* layout for an ext2 file system is shown in Figure 1. The file system is split into logical *block groups* to keep file blocks to be closer together on disk. The first 1024 bytes are reserved for boot data

(even if unused), followed by the super-block (also 1024 bytes). The super-block contains most of the file system information as a massive structure, part of which you can see below.

Ext2 Super Block Structure

```
struct ext2_super_block {
    uint32_t inode_count; /* Total number of inodes in the fs */
    uint32_t block_count; /* Total number of blocks in the fs */
    uint32_t reserved_block_count; /* Number of blocks reserved for root */
    uint32_t free_block_count; /* Number of free blocks */
    uint32_t free_inode_count; /* Number of free inodes */
    uint32_t first_data_block; /* The first data block number */
    uint32_t log_block_size; /* 2^(10 + this value) gives the block size */
    uint32_t log_fragment_size; /* Same for fragments (we won't use fragments) */
    uint32_t blocks_per_group; /* Number of blocks for each block group */
    uint32_t fragments_per_group; /* Same for fragments */
    uint32_t inodes_per_group; /* Number of inodes for each block group */
    uint32_t mount_time; /* Some less relevant fields */
    uint32_t write_time;
    uint16_t mount_count;
    uint16_t max_mount_count;
    uint16_t magic;
    uint16_t state;
    uint16_t errors;
    uint16_t minor_rev_level;
    uint32_t last_check_time;
    uint32_t check_interval;
    uint32_t creator_os;
    uint32_t rev_level; /* Revision level: 0 or 1 */
    uint16_t default_uid;
    uint16_t default_gid;
    uint32_t first_inode; /* First non-reserved inode in the file system */
    uint16_t inode_size; /* Size of each inode */
    /* More, less relevant fields follow */
};
```

- **Block group descriptor table blocks**: These blocks follow the previously mentioned information.
- **Block group descriptors**: They store information about each block group.
- **Information stored in block group descriptors**: This includes the positions of the bitmaps, inode table, and other details such as the number of free blocks in the block group.
  - **Bitmaps**: These mark allocated block and inode positions in the block group.
  - **Inode table**: It provides space for all the inodes in the block group and is static. The maximum number of inodes is fixed, and unallocated inodes are represented as sequences of zeroes in the table.
  - **Data blocks**: The remaining blocks are allocated for storing data to be used by files.
  - **Backups in some block groups**: Apart from the first block group, some block groups may also contain backups for the superblock and block group descriptor table.

- **Structure definition**: The structure definition of the file system is provided below the mentioned information.

Ext2 Block Group Descriptor Structure

```
struct ext2_block_group_descriptor {
    uint32_t block_bitmap; /* Block containing the block bitmap */
    uint32_t inode_bitmap; /* Block containing the inode bitmap */
    uint32_t inode_table; /* First block of the inode table */
    uint16_t free_block_count; /* Number of free blocks in the group */
    uint16_t free_inode_count; /* Number of free inodes in the group */
    uint16_t used_dirs_count; /* Number of directories in the group */
    uint16_t pad; /* Padding to 4 byte alignment */
    uint32_t reserved[3]; /* Unused, reserved 12 bytes */
};
```

- **File information stored in inodes**:
  - **Mode (type/permissions)**: Specifies the type of file and its permissions.
  - **Owner**: Identifies the owner of the file.
  - **Size**: Indicates the size of the file.
  - **Link count**: Represents the number of hard links to the file.
  - **Timestamps**: Stores timestamps related to the file, such as creation, modification, and access times.
  - **Attributes**: Additional attributes associated with the file.
- **Pointers to file data in the inode**:
  - **12 direct blocks**: Contains direct pointers to data blocks that store file content.
  - **Single indirect block**: Contains pointers to other data blocks indirectly, allowing for additional file data storage.
  - **Double indirect block**: Contains pointers to single indirect blocks, which in turn point to data blocks.
  - **Triple indirect block**: Contains pointers to double indirect blocks, which further point to single indirect blocks and then to data blocks.
- **Indirect blocks**:
  - Indirect blocks are data blocks filled with pointers.
  - They enable indirect indexing, allowing for multiple data blocks to be accessed through these pointers.

As an example, an ext2 file system with a block size of 4096 will be able to store $4096/4 = 1024$ block numbers in an indirect block. Thus, a single indirect block would be able to index $1024 \cdot 4096 = 4$MB of data. A double indirect block would index 1024 indirect blocks, indexing a total of $1024 \cdot 1024 \cdot 4096 = 4$GB of data. A triple indirect block would be able to index 4TB of data! The data can have *holes*, meaning that some intermediate blocks may not be allocated. You can see the inode structure below. [1]

[1] Although the maximum file size for a 4KB block ext2 file system would be around 2TB due to the 4-byte limit of the `i_blocks` field (named `block_count_512` in `ext2fs.h`) in the inode.

Ext2 Inode Structure

```
struct ext2_inode {
    uint16_t mode; /* Contains filetype and permissions */
    uint16_t uid; /* Owning user id */
    uint32_t size; /* Least significant 32-bits of file size in rev. 1 */
    uint32_t access_time; /* Timestamps (in seconds since 1 Jan 1970) */
    uint32_t creation_time;
    uint32_t modification_time;
    uint32_t deletion_time; /* Zero for non-deleted inodes! */
    uint16_t gid; /* Owning group id */
    uint16_t link_count; /* Number of hard links */
    uint32_t block_count_512; /* Number of 512-byte blocks alloc'd to file */
    uint32_t flags; /* Special flags */
    uint32_t reserved; /* 4 reserved bytes */
    uint32_t direct_blocks[12]; /* Direct data blocks */
    uint32_t single_indirect; /* Single indirect block */
    uint32_t double_indirect; /* Double indirect block */
    uint32_t triple_indirect; /* Triple indirect block */
    /* Other, less relevant fields follow */
};
```

Note that inodes do not contain file names. These are instead contained in directory entries referring to inodes. Thus, it's possible to have entries with different names in different directories referring to the same file (inode). Each of these is called hard links.

Directories are also files and thus have inodes and data blocks. However, data blocks of directories have a special structure. They are filled with directory entry structures forming a singly linked list:

Ext2 Directory Entry Structure

```
struct ext2_dir_entry {
    uint32_t inode; /* inode number of the file */
    uint16_t length; /* Record length, aligned on 4 bytes */
    uint8_t name_length; /* 255 is the maximum allowed name length */
    uint8_t file_type; /* Not used in rev. 0, file type identifier in rev. 1 */
    char name[]; /* File name. This is called a 'flexible array member' in C. */
};
```

These are records of variable size, essentially 8 bytes plus space for the name, aligned on a 4-byte boundary [2]. The next entry can be reached by adding `length` bytes to the current entry's offset. The last entry has its length padded to the size of the block so that the next entry corresponds to the offset of the end of the block. Once the end of the block is reached, it's time to move on to the next data block of the directory file. As one last thing, a 0 `inode` value indicates an entry which should be skipped (can be padding or pre-allocation).

Some important information before finishing up:

- The super block *always* begins at byte 1024.
- The BGD table is at the beginning of the first unoccupied block after the super block.

[2] Why? Remember your computer organization course!

- Inode numbering starts at 1. Not zero!
- The first block number depends on the block size and should be read from the superblock.
- The first 10 inodes are reserved for various purposes.
- Inode 2 is always the root directory inode.
- Inode 11 usually contains the `lost+found` directory under root.
- Bitmap and inode table offsets are not fixed. You should not assume anything. Read them from the block group descriptor.
- The last block group can have fewer blocks and inodes than indicated by the `*_per_group` fields, depending on the total.

The following links contain details about ext2 and should be your go-to references when writing your code:

- ext2 documentation by Dave Poirier: `http://www.nongnu.org/ext2-doc/ext2.html`
- OSDev wiki article on ext2: `https://wiki.osdev.org/Ext2`
- Another, more history-focused article from Dave Poirier: `http://web.mit.edu/tytso/www/linux/ext2intro.html`

2.2 Image Creation

To create an ext2 file system image, first, create a zero file via `dd`. Here's an example run creating a 512KB file (512 1024-byte blocks).

```
$ dd if=/dev/zero of=example.img bs=1024 count=512
```

Then, format the file into a file system image via `mke2fs`. The following example creates an ext2 file system with 64 inodes and a block size of 2048.

```
$ mke2fs -t ext2 -b 2048 -N 64 example.img
```

You can dump file system details with the `dumpe2fs` command:

```
$ dumpe2fs example.img
```

Now that you have a file system image, you can mount the image onto a directory. The FUSE-based `fuseext2` command allows you to do this in userspace without any superuser privileges (use this on the ineks! [3]). The below example mounts our example file system in read-write mode:

```
$ mkdir fs-root
$ fuseext2 -o rw+ example.img fs-root
```

[3]Note that **fusermount** does not currently work on the ineks without setting group read-execute permissions for the mount directory (`chmod g+rx`) and group execute permissions (`g+x`) for other directories on the path to the mount directory (including your home directory). This is not ideal and is being looked into so that it can work with user permissions only. An announcement will follow when a fix happens.

Now, `fs-root` will be the root of your file system image. You can `cd` to it, create files and directories, do whatever you want. To unmount once you are done, use `fusermount`:

```
$ fusermount -u fs-root
```

Make sure to unmount the file system before running programs on the image. On systems where you have superuser privileges, you can use the more standard `mount` and `unmount`:

```
$ sudo mount -o rw example.img fs-root
$ sudo umount fs-root
```

You can check the consistency of your file system after modifications with `e2fsck`. The below example forces the check-in verbose mode and refuses fixes (`-n` flag), since `e2fsck` attempts to fix problems by default. This will help you find bugs in your implementation later on.

```
$ e2fsck -fnv example.img
```

3 Implementation

You will write a file system integrity recovery program `recext2fs` that can fix ext2 images with certain kinds of damages, located explicitly at bitmaps and pointers to file data in inodes. All of the damage is the shape of bits set incorrectly to 0.

All other data in the filesystem (such as contents of data blocks, super block, and block group descriptor table) is intact, including the content of reserved inodes.

Regarding the contents of data blocks, all non-empty data blocks not used for directories start with the same 32 bytes that will be given to you in runtime so you can separate directories from user files. There is no corruption in pointers to empty data blocks, meaning all used data blocks are findable. Also, unused memory is wiped clean just before the corruption occurs, meaning there are no data blocks with data in them that are not currently being used. This is not the case for inode tables.

3.1 Bitmap Recovery (50 pts)

You need to check the integrity of bitmaps in the filesystem, both the block bitmaps and the inode bitmaps, and fix any bits that are incorrectly set to 0.

Information given in the initial file system is, by itself, enough to fix all bitmaps. You do not need to combine pointer recovery and bitmap recovery to get the full points from bitmap recovery. However, having an intact inode structure might make this process easier.

3.1.1 Inode Bitmap Recovery (20 pts)

Because inodes have set sizes and are located in predictable locations, checking all inodes to decide whether they are being used is easy. However, as stated before, these can vary between cases. You must gather this information through the super block and block group descriptors.

Also, unlike data blocks, unused inodes were not wiped, meaning the existence of data by itself is insufficient to determine whether an inode is being used.

3.1.2 Block Bitmap Recovery (30 pts)

Fixing a block bitmap is a more difficult procedure with several edge cases. The easiest fix possible is using the fact that the unused space was wiped and setting the bits corresponding to any block that contains data. But more is needed as there might be blocks in use that might not have any data. But these are guaranteed to be pointed to by an intact pointer. Intact pointers located in inodes are the best place to start, especially if you know which inodes are being used, but because of the damage to the file system, this also might not be enough for some cases. There might be empty data blocks being pointed by indirect pointers, but pointers to data blocks containing indirect pointers might not be intact. You need to distinguish between blocks with pointers, directory entries, and user data to resolve this.

3.2 Pointer Recovery (70 pts)

You must also check the integrity of data block pointers in the file system and fix any bits incorrectly set to 0. Only pointers located at inodes contain incorrectly set pointers; any data block that contains indirect pointers is completely intact.

Information given in the initial file system is, by itself, enough for fixing all pointers. You do not need to combine pointer recovery and bitmap recovery to get the full points from pointer recovery. However, having an intact bitmap structure might make this process easier.

This procedure is a lot harder and nearly entirely consists of edge cases. Most of the information needed to achieve this is located in the size field of the inodes, directory entries, and data blocks with pointers.

For directory entries, any data block with the directory entry "./" must be pointed to by the inode of that directory entry. Also, any data block with the directory entry "../" must be pointed to by an inode for a directory entry located within the contents of the "../" directory entry.

Unreferenced indirect pointer blocks can be matched with inodes that are missing them by comparing the number of blocks that are pointed by that data block and amount of missing data blocks required to achieve the file size recorded at the inode. To accomplish this correctly, you need to make sure that the data block with pointers is truly unreferenced (including indirect pointers) and determine if it is a single, double, or triple indirect block.

After you exhausted every methodology, there might be at most one non-empty data block remaining that contains user data that is not being pointed to by any inode. Because there is, at most, only one such data block, only one inode must be missing a pointer that fits this description. There can also be an unreferenced data block fitting this description that contains directory entries. Assuming you have paired everything else correctly, you can associate these blocks with the only remaining missing direct pointers.

If an inode has more than one direct pointers missing, you should place the missing pointers in ascending order. This does not mean direct pointers that are not missing will be in ascending order.

3.3 User Conformation (20 pts)

After fixing the image, you should print whole the directory structure of the file system as a tree (without depth limit) to the standard output, allowing the user to check the surface-level correctness of the file system.

To achieve this, you need to start from the root and print the contents of each directory in a depth-first manner. For example, in a file system where "root" is the root directory, "home" and "etc" are its children, and so on, a valid output might look like the following:

```
- root/
-- home/
```

```
--- user1/
---- file1.txt
---- file2.txt
--- user2/
---- file3.txt
-- etc/
--- config.txt
```

As it can be seen in the example, the name of each directory entry should be preceded by a number of "-" characters equal to the depth level of the entry in the directory hierarchy (root folder having the depth level of 1). The order of entries in a directory does not matter, but you must preserve the hierarchy. This means that for the previous directory hierarchy, the following is a valid output:

```
- root/
-- home/
--- user2/
---- file3.txt
--- user1/
---- file1.txt
---- file2.txt
-- etc/
--- config.txt
```

While this one is not:

```
- root/
-- home/
--- user1/
--- user2/
---- file1.txt
---- file2.txt
---- file3.txt
-- etc/
--- config.txt
```

3.4 Grading

Your program will be called from the command line with the following format:

```
$ ./recext2fs image_location data_identifier
```

where `image_location` is the location of the image relative to your executable, and `data_identifier` is the 32 bytes that is located at the start of each non-empty data block containing user file data.

You need to perform the necessary recovery procedures described in Sections 3.1 and 3.2 in place. You will read the image given by `image_location`, recover it, and save the resulting image to the same location. As you are not using the file system, you only need to change the corrupted spaces without updating any other fields. Doing unnecessary or wrong modifications to the file system will result in a reduction in your grades.

Then you will print the directory hierarchy to standard output as described in Section 3.3.

Not all test cases will include both bitmap and pointer corruption; some test cases might not even contain any corruption at all.

You will be graded according to how you modified the file system and your output, but each part will be graded separately for each test case.

While it will not be checked specifically, try not to have memory leaks. You will be working with very big files on inek machines, if any problems occurs due to memory leaks, you will get 0 for that case, regardless of any bits you have changed at the time of the problem.

Your code needs to end its execution within 5 minutes by test case. If it takes more than that, you will lose 1% of your grade for that test case per second. For especially large cases, meaning one of the following:

- `inode_count` > 20000
- `block_count` > 100000
- `log_block_size` > 2

This time limit will be extended to 10 minutes.

Not being compliant with the submission format and submitting late will result in a reduction in your grade, described in Section 6.

4 Specifications

1. Your code must be written in C or C++.
2. Your implementation will be compiled and evaluated on the `ineks`, so you should make sure that your code works on them.
3. For simplification, the outputs can be in any order.
4. You are supposed to read the file system data structures into memory, modify them, and write them back to the image. Mounting the file system in your code is forbidden; do not run any other executables from your code using things like `system()` or `exec*()`. This includes `mkfs.ext2j`, which is provided to journal an ext2 file system.
5. Including POSIX ext2/ext3/ext4 libraries (as well as kernel codes and their variations etc.) is **not** allowed.
6. The `ext2fs.h` header file is provided for your convenience. You are free to include it, modify it, remove it, or do whatever you want with it.
7. The `identifier.h` and `identifier.c` files are also provided for your convenience. You are free to include it, modify it, remove it, or do whatever you want with it.
8. Your submission should include every file you used in your code. No files will be supplied to you.
9. We have a zero-tolerance policy against cheating. All the code you submit must be **your own work**. Sharing code with your friends, using code from the internet or previous years' homework are all considered plagiarism and strictly forbidden.
10. Follow the course page on ODTUClass for possible updates and clarifications.
11. Please ask your questions on ODTUClass instead of sending an email for questions that do not contain code or solutions so that all may benefit.

5 Tips and Tricks

- You can use the number of free blocks/inodes contained in the block group descriptor to double-check your solutions.
- There is partial credit for each fix you make, and the type of fixes required are mostly independent. This means you can start with the parts you find the easiest and go step by step, skipping the parts you struggle with.
- While it is supposed to work after the file system is fixed, the user conformation part requires only the most fundamental of operations (which you probably have to implement for the first two parts), is easy to test, and does not contain edge cases, making it an ideal start point.
- It might be beneficial for you to track what points to a data block (which inode or indirect block) while implementing the pointer recovery process, but it is not required.

6 Submission

Submission will be done via ODTUClass. Create a gzipped tarball file named `hw3.tar.gz` that contains all your source code files together with your Makefile. Your archive file should not contain any subfolders. Your code should compile, and your executable should run with the following command sequence:

```
$ tar -xf hw3.tar.gz
$ make all
$ ./recext2fs image_location data_identifier
```

If there is a mistake in any of these steps mentioned above, you will lose 10 points.

Late Submission: a penalty of $5 \cdot (\text{late days})^2$ will be applied to your final grade.