1 Introduction

This final homework aims to familiarize you with the implementation of file systems. You will be required to implement various utilities on a simplified version of the ext2 file system, which is a traditional file system. Additionally, there is a bonus opportunity to extend the ext2 file system by incorporating a feature called "transactional metadata journaling". This feature helps back up disk transactions and reduces the risk of disk failures.

Detailed explanations of the journaling modification will be provided, along with some utilities to assist with image creation and checking. Your code will directly operate on the ext2 file system image, performing operations that update the file system. It's a challenging task ahead, so get ready to dive in!

2 ext2 (100 points)

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.

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.

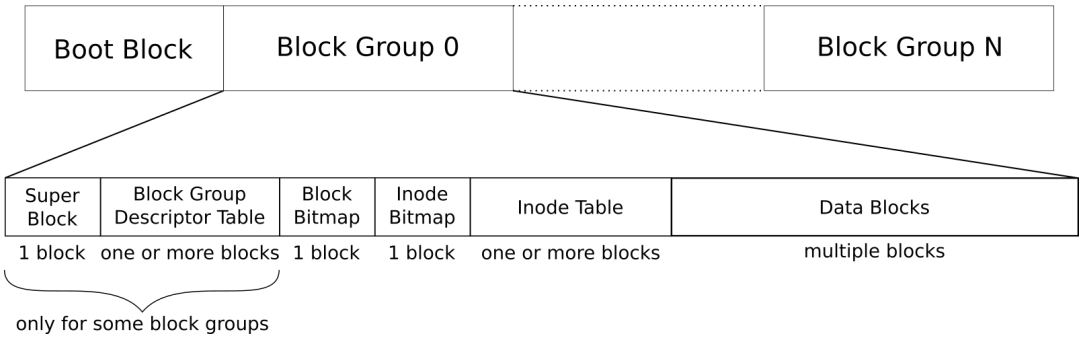

Figure 1: Overall ext2 Layout

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 $1024 \cdot 4096 = 4\text{MB}$ of data. A double indirect block would index 1024 indirect blocks, indexing a total of $1024 \cdot 1024 \cdot 4096 = 4\text{GB}$ 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]

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

[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.

[2] Why? Remember your computer organization course!

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.
- 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:

[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.

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

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
```

2.3 Implementation

You will write a utility program `je2fs` that will be able to perform some operations on the ext2 images. For the base part of the homework, you can assume that all utility functions you will write should perform the same as ext2 utility functions, you can cheat from them (:.

2.3.1 mkdir - creating an empty directory (20 pts)

The first utility you will write is `mkdir`, which creates an empty directory under the given path. The command will be run with the following format:

```
$ ./je2fs FS_IMAGE mkdir /abs/path/to/dirname
```

- `FS_IMAGE` is a valid ext2 image
- `/abs/path/to/dirname`: An absolute path to the file starting from the root directory "/" (specification 4). However, the separator is not guaranteed to be a single slash (specification 5).

Let's consider a directory creation scenario: There is a directory, under the given **path**, having no entry named **"directory"** only. Let's say the smallest unallocated inode index is **15**, and the smallest unallocated block index **585**. Additionally, the directory under **path**, has a single block at index **540** to store its entries and there is enough space to store the entry named **"directory"**.

Therefore in this scenario, we would expect to see such a case:

```
$ ./je2fs FS_IMAGE mkdir /abs/path/to/directory
EXT2_ALLOC_INODE 15
EXT2_ALLOC_BLOCK 585
EXT2_ALLOC_DIR_ENTRY "directory" 540
EXT2_ALLOC_DIR_ENTRY "." 585
EXT2_ALLOC_DIR_ENTRY ".." 585
```

The corresponding algorithm to be executed will be like below

Ext2 Create Directory Algorithm

```
procedure EXT2_MKDIR(path="/path/to/dirname")
    parent_path, name ← tokenize the path
    if (/abs/path/to/ does not exists) or (/abs/path/to/dirname exists) then
        return
    end if
    parent_inode ← get inode of the parent_path

    // create a new directory inode
    inode ← allocate a new inode
    block ← allocate a new block
    link the block to the inode

    // link parent_inode -> inode with name
    create a dir entry with name under parent_inode
    link the created dir entry with name and inode

    // link inode -> inode with "."
    create a dir entry with "." under inode
    link the created dir entry and inode

    // link inode -> parent_inode with ".."
    create a dir entry with ".." under inode
    link the created dir entry and parent_inode

    // update time and write them back
    reverse through parent_inode to ROOT_INODE
    update their modification and access times
end procedure
```

2.3.2 rmdir - removing an empty directory (20 pts)

The second utility you will write is `rmdir`, which removes an empty directory under the given path. The command will be run with the following format:

```
$ ./je2fs FS_IMAGE rmdir /abs/path/to/directory
```

Where you can assume the followings:

- `FS_IMAGE` is a valid ext2 image
- `/abs/path/to/dirname`: An absolute path to the file starting from the root directory "`/`" (specification 4). However, the separator is not guaranteed to be a single slash (specification 5).

The corresponding algorithm to be executed is like below:

Ext2 Remove Directory Algorithm

procedure EXT2_RMDIR(**path**="/path/to/dirname")
  parent_path, **name** ← tokenize the **path**
  if (`/abs/path/to/dirname` does not exists) **or** (`/abs/path/to/dirname` has more than two dir entry) **then**
    return
  end if
  parent_inode ← get inode of the **parent_path**
  inode ← get inode of the **path**
  // unlink parent_inode –> inode with **name**
  remove a dir entry with **name** under **parent_inode**
  unlink the created dir entry with **name** and **inode**
  // unlink dir entries under **inode**
  decrement **inode** link count by two
  // make inode's blocks and inode itself free again
  for each **block** in **inode**'s direct and indirect blocks **do**
    // i didn't say **unlink** :)
    deallocate the **block**
  end for
  deallocate the **inode**
  // update time and write them back
  reverse through **parent_inode** to **ROOT_INODE**
  update their modification and access times
end procedure

Let's consider a scenario: There is a directory, under the given **path**, having tow entry named **"."** and **".."** only. Let's say the directory's inode index is **15**, and two entries are located in the block index **585**. Additionally, the directory name is an entry in the block index **540** which is linked to the parent inode.

Therefore in this scenario, we would expect to see such a case:

```
$ ./je2fs FS_IMAGE rmdir path
EXT2_FREE_DIR_ENTRY "directory" 540
EXT2_FREE_BLOCK 585
EXT2_FREE_INODE 15
```

For simplification, you are free to output them in any order.

2.3.3 rm - removing a regular file (10 pts)

The third utility you will write is `rm`, which removes a regular file with the given name. The format for this command is

The command will be run with the following format:

```
$ ./je2fs FS_IMAGE rm /abs/path/to/file
```

Where you can assume the followings:

- `FS_IMAGE` is a valid ext2 image

- `/abs/path/to/dirname`: An absolute path to the file starting from the root directory "`/`" (specification 4). However, the separator is not guaranteed to be a single slash (specification 5).

The corresponding algorithm you will write will be below:

Ext2 Remove File Algorithm

```
procedure EXT2_READ_FILE(path="/path/to/fname")
    parent_path, fname ← tokenize the path
    if /abs/path/to/fname does not exists then
        return
    end if
    parent_inode ← get inode of the parent_path
    inode ← get inode of the path
    // remove the link (parent_inode -> inode) with fname
    remove a dir entry with fname under parent_inode
    unlink the created dir entry with fname and inode
    for each block in inode's direct and indirect blocks do
        deallocate the block
    end for
    if inode's link_count is zero then
        deallocate the inode
    end if
    // update time and write them back
    reverse through parent_inode to ROOT_INODE
    update their access times
end procedure
```

Let's consider a scenario: There is a file, under the given **path**, whose content is among the blocks **555**, **556**, and **557**. Let's say the file's inode index is **15**, and the file name is in an entry at the block index **540** which is linked to the parent inode. Additionally, assume that there is no hard link to the file.

Therefore in this scenario, we would expect to see such a case:

```
$ ./je2fs FS_IMAGE rm path
EXT2_FREE_DIR_ENTRY "file name" 540
EXT2_FREE_BLOCK 555
EXT2_FREE_BLOCK 556
EXT2_FREE_BLOCK 557
EXT2_FREE_INODE 15
```

2.3.4 ed - modifying a regular file (50 pts)

The last utility you will write is `ed`, which replaces the given **string** to a regular file with the given **name**. The format for this command is

The command will be run with the following format:

```
$ ./je2fs FS_IMAGE ed -i INDEX -b BACKSPACE /abs/path/to/file -- STRING
```

Where you can assume the followings:

- `FS_IMAGE` is a valid ext2 image

- `INDEX` is the offset in the file that you start insertion.
  - you can assume that **INDEX** is less then or equal to the **STRING** length.
- `BACKSPACE` is the number of characters to be deleted at INDEX before insertion.
  - you can assume that **BACKSPACE** is less than or equal to **INDEX**.
- `/abs/path/to/dirname`: An absolute path to the file starting from the root directory "`/`" (specification 4). However, the separator is not guaranteed to be a single slash (specification 5).
- `STRING` is enclosed by single quotes so that you don't have to deal with joining multiple argv's.

For our examples, let's say there is a file, under the given **path**,

- which contains **LENGTH** ($= 3 * \textbf{BLOCK SIZE} - 12$) number of characters as its content.
- which contains blocks **555**, **556**, and **557**,
- whose inode index is **15**
- whose file entry is at the block **540** which is contained by the parent directory.

Additionally assume that we have an input **string** whose length is **STRLEN** = **BLOCK SIZE** + 20, where **BLOCK SIZE** > 20. Let's consider an insert-mode example:

```
$ ./je2fs FS_IMAGE ed -i INDEX -b 0 path -- string
EXT2_ALLOC_BLOCK 558
EXT2_ALLOC_BLOCK 559
```

This is simply an insert command. We will start at **INDEX**, insert **string**, and append the remaining content after **INDEX** to the file. Since $4 * \textbf{BLOCK SIZE} < 3 * \textbf{BLOCK SIZE} - 12 + \textbf{BLOCK SIZE} + 20 < 5 * \textbf{BLOCK SIZE}$, this command will allocate two new blocks which are **558** and **559**.

Let's consider a replace-mode example:

```
$ ./je2fs FS_IMAGE ed -i INDEX -b STRLEN path -- string
```

This is simply a replacement command. We will start at **INDEX**, delete **BLOCK SIZE** $+ 20$ characters, and insert **string** which is also **BLOCK SIZE** $+ 20$ characters. Therefore this command will only modify times in terms of inode metadata. Let's consider a more general example:

```
$ ./je2fs FS_IMAGE ed -i INDEX -b BLOCK_SIZE + 15 path -- string
EXT2_FREE_BLOCK 556
EXT2_FREE_BLOCK 557
```

In this case, we will start at **INDEX**, and delete **BLOCK SIZE** + **15** characters. We will then insert the **string**, and append the remaining content after **INDEX** to the file. Since $2 * \textbf{BLOCK SIZE} < (3 * \textbf{BLOCK SIZE} - 12) + (\textbf{BLOCK SIZE} + 20) - (\textbf{BLOCK SIZE} + 15) < 3 * \textbf{BLOCK SIZE}$, this command will deallocate two blocks which are **556** and **557**.

The corresponding algorithm you will write will be below:

Ext2 Edit File Algorithm

```
procedure EXT2_EDIT_FILE(path="/path/to/file",content=STRING, index, bspace)
    parent_path, fname ← tokenize the path
    if /abs/path/to/fname does not exists then
        return
    end if
    parent_inode ← get inode of the parent_path
    inode ← get inode of the path

    // enqueue a buffer array with content + file[index:len(file)]
    content_buffer ← Queue(content)
    for each block in in inode's direct and indirect blocks do
        keep track of an offset indicating your location in the file by character
        if index < offset then
            enqueue from the content in the block at offset till the end of block
        end if
    end for

    // dequeue the buffer array into file[index - bspace:len(file)]
    for each block in in inode's direct and indirect blocks do
        if content_buffer is not empty then
            keep track of an offset indicating your location in the file by character
            if index - bward < offset then
                dequeue at most (end of block - offset) characters to the block starting at offset
            end if
        else
            if offset < the start position of the block then
                deallocate the block
            end if
        end if
    end for

    // if buffer array is not empty then dequeue it to new blocks
    block ← the last block of inode
    while content_buffer is not empty do
        keep track of an offset indicating your location in the file by character
        if offset == end of block then
            block ← allocate a block
        end if
        dequeue to the block starting at the offset
    end while

    // update time and write them back
    reverse through parent_inode to ROOT_INODE
    update their access and modification times
end procedure
```

3 ext2journal (bonus 50 points)

3.1 Details

Now that the basics of ext2 are out of the way, it's time to *extend* the second *extended* file system! What we want is the ability to journal the metadata of inodes and blocks, so that we can have atomic-like operations on disk to prevent inconsistencies in case of any power failure.

Thankfully, Stephen Tweedie has developed a journaling extension for the ext2 code, providing several benefits. One major advantage is the prevention of metadata corruption and the elimination of the need to wait for `e2fsck` to complete after a system crash. Importantly, this extension can be implemented without modifying the existing on-disk ext2 layout. The journal itself functions as a regular file, storing modified metadata blocks (and optionally data blocks) before they are written to the file system. This approach allows for the addition of a journal to an ext2 file system without requiring any data conversion.

When file system changes occur, such as file renaming, they are recorded as transactions within the journal. These transactions can be either complete or incomplete at the time of a system crash. In the case of a complete transaction, where the system does not crash, the blocks within that transaction are guaranteed to represent a valid state of the file system and are subsequently copied into the file system. However, if a transaction is incomplete at the time of a crash, the blocks within that transaction lack consistency guarantees and are therefore discarded. As a result, any file system changes represented by those discarded blocks are lost.

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

- Journaling Extension in EXT2: `https://www.kernel.org/doc/html/latest/filesystems/ext2.html#journaling`
- Journal Desing in LinuxExpo98: `https://pdos.csail.mit.edu/6.828/2020/readings/journal-design.pdf`

EXT2 Journal Metadata Block Structure

```
struct ext2_journal_metadata_block {
    uint8_t data[]; /* entire contents of a single block of file system metadata */
};
```

A journal metadata block contains the entire contents of a single block of file system metadata as updated by a transaction. This means that however small a change we make to a file system metadata block, we have to write an entire journal block out to log the change. However, this turns out to be relatively cheap for two reasons/

EXT2 Journal Descriptor Block Structure

```
struct ext2_journal_descriptor_block {
    /* number of metadata blocks to be written */
    uint32_t metadata_blocks_count;
    /* Disk block numbers of the metadata blocks */
    uint32_t metadata_blocks_array[EXT2_MAX_METADATA_BLOCKS];
};
```

Descriptor blocks are journal blocks which describe other journal metadata blocks. Whenever we want to write out metadata blocks to the journal, we need to record which disk blocks the metadata normally lives

at, so that the recovery mechanism can copy the metadata back into the main file system. A descriptor block is written out before each set of metadata blocks in the journal and contains the number of metadata blocks to be written plus their disk block numbers.

Both descriptor and metadata blocks are written sequentially to the journal, starting again from the start of the journal whenever we run off the end. At all times, we maintain the current head of the log (the block number of the last block written) and the tail (the oldest block in the log which has not been unpinned, as described below). Whenever we run out of log space –the head of the log has looped back around and caught up with the tail– we stall new log writes until the tail of the log has been cleaned up to free more space.

EXT2 Journal Header Block Structure

```
struct ext2_journal_header_block {
    /* sequence number */
    uint64_t sequence_number;
    /* current head of the journal (block number of the last block written) */
    uint32_t current_head;
    /* oldest block in the log which has not been unpinned (tail) */
    uint32_t tail;
};
```

Finally, the journal file contains a number of header blocks at fixed locations. These record the current head and tail of the journal, plus a sequence number. At recovery time, the header blocks are scanned to find the block with the highest sequence number, and when we scan the log during recovery we just run through all journal blocks from the tail to the head, as recorded in that header block.

3.2 Image Creation

Thankfully, your implementations will work on ext2 images, and to make it ext2-compatible journaled we need to only create a file named ‘‘`/header.journal`’’ with inode 12. Jorunal file will have two blocks. The second block is an empty descriptor file, therefore its count is zero. The first block is a header block, whose head and tail are referring to the second block index.

This can be simply done by creating an empty file on a fresh ext2 image.

```
$ ./mkfs.ext2j example.img
```

The `mkfs.ext2j` executable[4] compiled for x86_64 Linux is provided to help you deal with ext2s images. Rather than creating images from scratch, it converts existing ext2 images into the journaled ext2 by allocating an inode at a specific index for the journal file.

Note that this will *only* create a journal file whose inode is indexed at 12. For checking general consistency, you should continue to use `e2fsck`. It should not report any problems after conversion (it’s a bug in mkfs.ext2j if it does).

Important: *Programming correctly* is a difficult art to master, and as such `mkfs.ext2j` may contain some bugs. If you suspect a bug, save the command and image reproducing the bug and send them over to adhd@ceng.metu.edu.tr.

[4]Because it’s going to be your sidekick! Also, anthropomorphizing your executables will help you feel less lonely when coding long stuff :) What? No, no, I’m totally fine!

The most recent version of mkfs.ext2j will always be available at `https://user.ceng.metu.edu.tr/~adhd/334hw3/mkfs.ext2j` along with a changelog at `https://user.ceng.metu.edu.tr/~adhd/334hw3/changelog.txt`. Please check the changelog for known bugs and possible bug fixes if you have issues and update your executable.

See the appendix for more details about the tool and processes.

3.3 Implementation

In this part also, you will be still writing to the utility program `je2fs` that will be able to perform some operations on the ext2 images. Apart from the base part of the homework, you may assume that your code will be recompiled with an extra option **-DJOURNAL** during compilation.

In terms of journaling, your functions are expected to act the same as the previous part, in terms of outputting and end-user behaviour. The only main difference is to use a small portion of the disk (which is called journal blocks or the journal file) as a buffer or a cache. That is, your journaled functions are basically expected to seek the journal first before reading (part 1) and writing (part 2) until a **commit** is received. Once it is received, changes in the journal file will be reflected (part 3) to the remaining disk, which is called permanent data blocks. For simplification, you can assume that there will be exactly one journal file whose inode is set in the header file.

3.3.1 Reading from the journal (10 pts)

In order to read from the journal blocks in a transactional metadata journaling file system, you will need to traverse the journal file and extract the necessary information using the provided struct definitions.

First, you can read the header blocks at fixed locations to obtain the current head and tail of the journal. Starting from the tail block, you can iterate through the journal blocks using the sequence number recorded in the header block. For each journal block, you can check its type (metadata or descriptor) and accordingly interpret the contents using the defined struct formats. By printing the relevant details, such as metadata block content or disk block numbers, you can display the entire sequence of changes made to the file system. Remember to handle the case when the head of the log has looped back around and caught up with the tail, indicating that the log space has been reused. This process will give you a comprehensive view of the journaling activity within the file system.

In order to test if your executable is reading from the journal, the following two easy commands will be used to check inode metadata and dir entry blocks.

```
$ ./je2fs FS_IMAGE inode INODE_INDEX
```

In the first command, you are expected to fetch the inode from either the journal or the permanent blocks and print.

```
$ ./je2fs FS_IMAGE entry BLOCK_INDEX
```

In the second command, you are expected to calculate the permanent data block's actual block index in the disk given as **BLOCK INDEX**. Then, print all entries in the actual index. Corresponding **print_inode** and **print_dir_entry** functions will be given as utility functions.

The corresponding algorithm you will write will be below:

Ext2 Read Algorithm

```
procedure EXT2_READ(off_t offset, void *mem, size_t num, bool journal)
    if is not journaled then
        copy num bytes from the ext2 image starting at offset to the mem
        return
    end if

    block_index ← floor(offset / block size)
    for each descriptor block from head to tail do
        for each metadata block in the descriptor block's array do
            if metadata block is a journal block for the permanent block at block index then
                // read from the metadata journal block instead of the permanent block
                offset ← (metadata block's offset) + (offset mod block size)
                copy num bytes from the ext2 image starting at offset to the mem
                return
            end if
        end for
    end for

    // if there is no metadata journal block for the block at block index yet
    if the descriptor block at the header block.head is full then
        copy num bytes from the ext2 image starting at offset to the mem
    end if
end procedure
```

3.3.2 writing to the journal (20 pts)

To implement journaling for permanent data in a transactional metadata journaling file system, you can utilize the provided struct definitions effectively. This includes both journaling directory metadata and regular file metadata, as they are considered permanent data within the file system. Whenever a transaction modifies directory metadata or regular file metadata, it is crucial to log these changes in the journal to ensure data consistency.

By journaling permanent data in this manner, including both directory metadata and regular file metadata, you ensure that modifications to these critical components of the file system are durably logged. In case of failures or system crashes, the recovery process can rely on the journal to restore the permanent data blocks to their latest consistent state. This approach guarantees data integrity and enables the transactional metadata journaling file system to reliably handle modifications to permanent data.

To test your implementation in this part, the commands in the previous part 2 will be used again. Those commands will be done on a fresh ext2 image with a journal. Then the journal file in the image will be wiped out, to compare if it has any extra modification on permanent data blocks, also the consistency of the journal file will be tested.

The corresponding algorithm you will write will be below:

Ext2 Write Algorithm

```
procedure EXT2_WRITE(off_t offset, const void *mem, size_t num, bool journal)
    if is not journaled then
        copy num bytes from the mem to the ext2 image starting at offset
        return
    end if

    block_index ← floor(offset / block size)
    for each descriptor block from head to tail do
        for each metadata block in the descriptor block's array do
            if metadata block is a journal block for the permanent block at block index then
                // update the existing metadata journal block instead of the permanent block
                offset ← (metadata block's offset) + (offset modulus block size)
                copy num bytes from the mem to the ext2 image starting at offset
                return
            end if
        end for
    end for

    // if there is no metadata journal block for the block at block index yet
    if the descriptor block at the header block.head is full then
        descriptor ← allocate and link a new block
        push descriptor block's index as the current head of the header journal block
    else
        descriptor ← seek the current head of the header journal block
    end if
    descriptor.array[descriptor.count++] ← allocate and link a new block
    offset ← (the new block's index * block size) + (offset modulus block size)
    copy num bytes from the mem to the ext2 image starting at offset
end procedure
```

3.3.3 commit - reflecting the changes (15 pts)

When recovering from a journal file in a transactional metadata journaling file system, the provided struct definitions will assist you in reconstructing the main file system. Begin by scanning the header blocks to find the block with the highest sequence number. This will give you the starting point for scanning the journal blocks during recovery. Starting from the tail block and moving towards the head block as recorded in the header block, iterate through the journal blocks. For each block, check its type (metadata or descriptor) and use the information stored in the descriptors to identify the metadata blocks that need to be restored. Retrieve the metadata block content and the disk block numbers from the descriptors and copy them back to their original locations in the main file system. By following this process, you can ensure that all changes recorded in the journal are applied to recover the file system to its latest consistent state. Handle cases where the head of the log has looped back and caught up with the tail, indicating that the log space needs to be cleaned up to free more space before continuing the recovery process

To test your implementation in this part, the following **commit** command will be executed. In terms of simulating possible crash scenarios, it is guaranteed that the journal file won't be changed. But you can imagine that we could prevent such cases by adding a **checksum** array to each descriptor block (by truncating the half of the metadata_block_array) for the corresponding journal metadata block and also an additional checksum for the descriptor block itself.

```
$ ./je2fs FS_IMAGE commit
```

During the evaluation, you can also consider that images from the previous read and write part will be subjected to noise (randomly bit-flipping), and we will check if the commit command recovers the image back to its original version just before the noise.

The corresponding algorithm you will write will be below:

Ext2 Commit Algorithm

procedure EXT2_COMMIT(off_t **offset**, void ***mem**, size_t **num**)
  while in the **header block**'s stack is not empty **do**
    seek the current **descriptor block index** in the **header file**
    read **descriptor block** from the ext2 image at **descriptor block index**
    while the **descriptor block**.count is positive **do**
      descriptor block.count ← **descriptor block**.count - 1
      metadata block index ← **descriptor block index** + **descriptor block**.count
      permanent block index ← **descriptor block**.array[**descriptor block**.count]
      read the journal block at **metadata block index**
      write to the block at **permanent block index**
    end while
    write **descriptor block** to the ext2 image at **descriptor block index**
    pop **descriptor block index** from the **header block**
  end while
end procedure

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. it is guaranteed that all paths are absolute.
5. However, the separator is not guaranteed to be a single slash. Therefore, following paths are valid:
   - /////abs/path///to///filename
   - //////////////dirname//////
6. 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.
7. Including POSIX ext2/ext3/ext4 libraries (as well as kernel codes and their variations etc.) is **not** allowed.
8. 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.

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 and COW for possible updates and clarifications.
11. Please ask your questions on COW instead of sending an email for questions that do not contain code or solutions, so that all may benefit.

5 Tips and Tricks

- The given steps for implementing `mkdir`, `rmdir`, `rm`, and `ed` (including bonus part `inode`, `entry`, and `commit`) are meant to guide you, but do not contain every single detail. As an example, the free block counts in the superblock and block group have to be updated when one is allocated or deallocated. You have to find out and take care of these details. `e2fsck` should help with this.
- Loading the file system image into memory is an excellent candidate for using `mmap`.
- Making seek, read, and write system calls in the custom reader and writer only. And using those custom commands in your solution will be a time-saver for you while migrating from section 2 to section 3.
- There are many operations involving traversing the blocks of an inode. Trying to come up with a configurable generic method to traverse them would be helpful, especially if you want to support indirect blocks.
- The data structures contain many unsigned 32-bit and 16-bit fields. Be very careful with overflows and negative values. Such bugs are hard to find and fix.
- Always check the consistency of the file system after modifying images to catch problems early. Back up your images.

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
$ ./je2fs # testing the base part
$ make all JOURNAL=true
$ ./je2fs # testing the bonus part
```

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.

7 Appendix - More ext2 Journal Details

7.1 Journal Block Types

There are essentially three *types* of journaling blocks that are valid in the transactional metadata journaling ext2 filesystem:

- **metadata block:** refers to a single block of filesystem metadata that is updated by a transaction. This can include information such as file permissions, ownership, timestamps, and other attributes associated with files and directories on the filesystem. When changes are made to a metadata block, the entire contents of the block must be written out to the journal in order to log the change. This ensures that any updates are recorded in case of an unexpected shutdown or power failure, allowing for quick and reliable recovery without data loss.
- **descriptor block:** refers to a journal block that describes other journal metadata blocks. Whenever metadata blocks are written out to the journal, we need to record which disk blocks the metadata normally lives at, so that the recovery mechanism can copy the metadata back into the main filesystem. A descriptor block is written out before each set of metadata blocks in the journal and contains the number of metadata blocks to be written plus their disk block numbers. The descriptor block is used by ext2fs during recovery to locate and restore the associated metadata blocks back into the main filesystem.
- **the journal file:** contains a number of header blocks at fixed locations. These record the current head and tail of the journal, plus a sequence number. The header block is used by ext2fs to keep track of the state of the journal and ensure that it remains consistent after a crash or power failure. The header block is also used to locate and read other blocks in the journal, such as descriptor blocks and metadata blocks.