Middle East Technical University | Department of Computer Engineering

CENG 334

Operating Systems

Spring 2020-2021

Homework 3 - Sharing Blocks in an ext2 Filesystem

Due date: 27 June 2021, Sunday, 23:59

1 Introduction

The objective in this final homework is getting familiar with how filesystems can be implemented. To this end, you will implement some utilities on our own extended version of the ext2 filesystem: ext2shared (ext2s), which allows blocks to be shared between different files.

The block sharing modification will be explained in detail, and some utilities to help with image creation and checking will be provided. Your code will work directly on this modified file system (ext2s), performing operations that will update the filesystem. Long road ahead, time to buckle up!

2 ext2

2.1 Filesystem Details

ext2 is a filesystem for Linux introduced in 1993, designed following older Unix filesystem principles. As a result, you will find topics covered in the course under UFS to be very relevant. Details of ext2 are at the core of this homework: knowing them is very important to both understanding how the extension will work and how to write your code. Newer versions of Linux use the ext4 filesystem. However, ext3 and ext4 are mainly extensions adding journaling and modern features. The core ideas and data structures remain the same as ext2. Structure definitions are provided for your use in the `ext2fs.h` header file and some are also shown below.

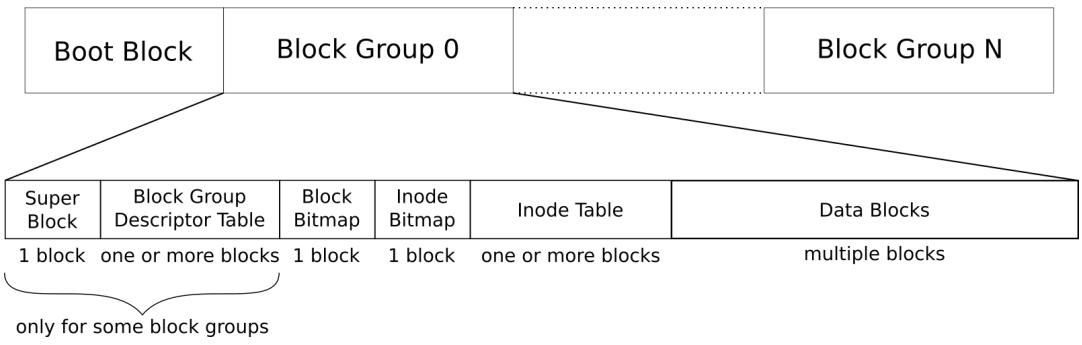

Figure 1: Example ext2 Layout

An *example* layout for an ext2 filesystem is shown in Figure 1. The filesystem is split into a bunch of logical *block groups* to encourage file blocks to be closer together on disk. The first 1024 bytes are reserved for boot data (even if unused), which is followed by the superblock (also 1024 bytes). The superblock contains most of the filesystem information as a huge 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 filesystem */
    uint16_t inode_size; /* Size of each inode */
    /* More, less relevant fields follow */
};
```

Then, block group descriptor table blocks follow. The table contains information about each block group stored in *block group descriptors*. These contain data about positions of the bitmaps, inode table as well as other things like the number of free blocks in the block group. The bitmaps simply mark allocated block and inode positions in the block group. The inode table contains space for all the inodes in the block group and is static: the maximum number of inodes is fixed and unallocated inodes reside in the table as sequences of zeroes. The rest of the blocks are data blocks to be used by files. Some block groups apart from the first also contain backups for the superblock and block group descriptor table, but most do not. The structure definition is shown below.

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 */
};
```

Inodes store file information: mode (type/permissions), owner, size, link count, timestamps and attributes are all stored in the inode. The inode also contains pointers to file data in the form of 12 direct blocks, one single indirect block, one double indirect block and one triple indirect block. Indirect blocks are data blocks filled with pointers to other blocks, which allow for many data blocks to be indexed indirectly. As an example, an ext2 filesystem 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]

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 multiple different entries having different names in multiple different directories referring to the same file (inode). Each of these are called hard links.

Directories are also files and thus have inodes and data blocks. However, data blocks of directories have

[1] Although the maximum file size for a 4KB block ext2 filesystem 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.

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, its 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.
- Inode numbering starts at 1. Not zero!
- The first block number depends on the block size and should be read from the super block.
- 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 lots of 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] Why? Remember your computer organization course!

2.2 Image Creation

To create an ext2 filesystem 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 filesystem image via `mke2fs`. The following example creates an ext2 filesystem with 64 inodes and a block size of 2048.

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

You can dump filesystem details with the `dumpe2fs` command:

```
$ dumpe2fs example.img
```

Now that you have a filesystem 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 filesystem in read-write mode:

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

Now, `fs-root` will be the root of your filesystem 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 filesystem 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 filesystem 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] 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.

3 ext2shared

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 share blocks between *different* files, so that we can duplicate huge files with zero cost and perhaps later on only allocate blocks that are changed with a copy-on-write mechanism. This would allow us to keep many slightly modified versions of large files with little extra space. ext2 has hard links, but does not support sharing blocks between files by default. The infrastructure is insufficient: it would be impossible to know when to de-allocate a block.

Thankfully, only a simple upgrade to the current mechanism is necessary. We need to count the number of references to blocks and only de-allocate them when the reference count reaches zero. Storing reference counts with the blocks themselves would complicate data management, so instead we want to create a block reference count table for each block group. This is pretty much analogous to the block bitmap. If we make our reference counters four bytes each (allowing up to around 4 billion references!), we will need 32 contiguous blocks per group to store the table in (since we will add a 32-bit counter for each bit in the block bitmap). The reference count table in each block group shall be called *refmap* from now on.

Now, where to put our refmaps? We could replace the block bitmaps with refmaps, since a positive reference count implies an allocated block, but that has a serious disadvantage. It would break backward compatibility and make us unable to use programs that work on ext2. If we instead keep the bitmap and include the refmap additively, we can keep using ext2 programs and drivers on ext2s; even though the reference counts will have to be updated separately.

Since each block group descriptor has 12 reserved, unused bytes, we can take 4 of those bytes and store the starting block of the refmap in it [4]. Time for a slight modification to our block group descriptor structure:

Ext2Shared 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 block_refmap; /* Sneaky time (: */
    uint32_t reserved[2]; /* 8 bytes still reserved */
};
```

The `block_refmap` field stores the first of the 32 contiguous blocks of the refmap, just like the `*_bitmap` and `inode_table` fields. The previously shown example layout in Figure 1 now becomes like Figure 2 with the addition of the refmap. Please note once again that the figure is an example, the position of the refmap is not guaranteed and should be read from the group descriptor.

3.2 Image Creation

Your implementations will work on ext2s images, and for that you need to be able to create ext2s images somehow for testing...

[4] *Reserved* means "Feel free to do whatever you want with it." as far as I'm concerned! Right?!

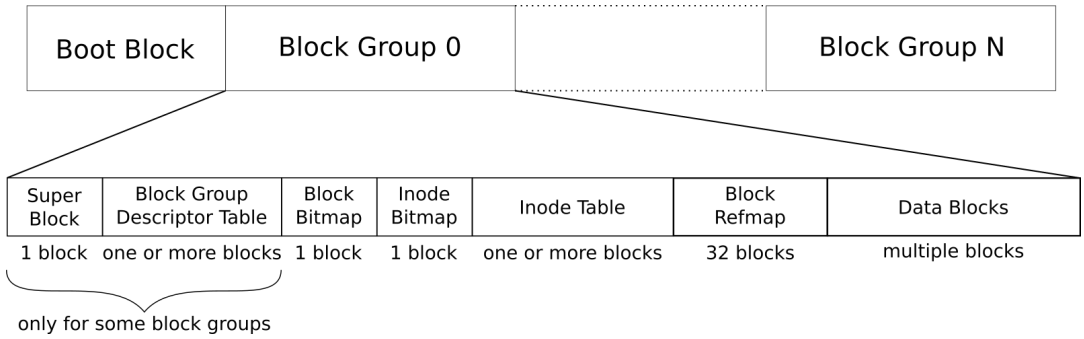

Figure 2: Example ext2s Layout

The `robin` executable[5] 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 ext2s by allocating refmaps:

```
$ ./robin convert example.img
```

This will allocate refmaps for each block group and set their offset in the descriptors. The minor revision level will be set to 334 and all the refmaps will also become available in a special file `".block_refmap"`. All of the blog group refmaps will appear contiguously as a huge block refmap under this file. Do note that conversion can fail if there is not enough space on your ext2 filesystem.

If you later re-mount your ext2s image as ext2 and modify it by adding/removing some files, you can update its refmaps afterwards as well:

```
$ ./robin update example.img
```

There's also a command for checking the consistency of the reference counts:

```
$ ./robin check example.img
```

Note that this will *only* check the reference counts and should be run on an otherwise consistent filesystem. For checking general consistency, you should continue to use `e2fsck`. It should not report any problems after conversion (it's a bug in robin if it does), but will start complaining about "multiply-claimed blocks" once blocks start being shared between files. This is normal; however you should keep your eyes open for different types of errors and also check the multiply-claimed blocks to make sure you are sharing the blocks you intended to share and not anything else.

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

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

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

[5]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!

4 Implementation

You will write a utility program `ext2sutils` that will be able to perform some operations on ext2s images. For the base part of the homework, you can assume that all the files (including directories) will be small and only include direct blocks and limit your implementation to that. That's it! Let's get right into it.

4.1 dup - duplicating files (60 pts)

The first utility you will write is `dup`, which is very similar to `cp` but will duplicate file blocks instead of copying data to new blocks. A simplified visualization of a duplicated inode is shown in Figure 3.

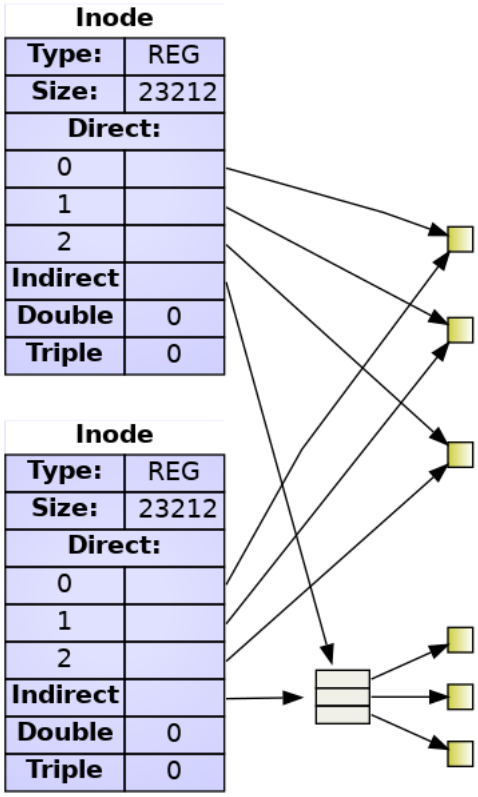

Figure 3: Simplified Duplicate Inode Block Layout

The command will be run with the following format:

```
$ ./ext2sutils dup FS_IMAGE SOURCE DEST
```

`FS_IMAGE` will be a valid ext2s filesystem image.

`SOURCE` always refers to *regular files* and has two possible formats:

1. `inode_no`: Just a valid inode number. With this format, `SOURCE` is the inode number of the file to be duplicated.
2. `/abs/path/to/filename`: An absolute path to the file starting from the root directory `/`. To simplify your implementation, the separator is guaranteed to be a single slash, there will never be multiple consecutive slashes.

`DEST` also has two possible formats:

1. `dir_inode/target_name`: The left side of the `/` is a directory inode, and the right side is the name of the destination duplicated file.
2. `/abs/path/to/target_name`: An absolute path, same properties as with `SOURCE`. The last component of the path is always the name of destination file and never just a directory. Intermediate directories in the path will always exist.

Some examples for extra clarification:

```
# dup file at inode 15 into directory inode 2 (root) with name file.txt
$ ./ext2sutils dup fs.img 15 2/file.txt
# dup file hw3.c under /folder/ into directory inode 14 with name code.c
$ ./ext2sutils dup myext2.img /folder/hw3.c 14/code.c
# dup file at inode 88 into directory /home/torag/docs/ with name recovered.bin
$ ./ext2sutils dup example.img 88 /home/torag/docs/recovered.bin
# dup file hr.avi under directory /y/2020 into /y/2021 with name nostalgia.avi
$ ./ext2sutils dup example.img /y/2020/hr.avi /y/2021/nostalgia.avi
```

To duplicate the file you need to:

1. In case there are absolute paths, determine the inode of the source file and target directory via path traversal. Otherwise, they're already given.
2. Allocate an inode for the new file.
3. Duplicate the metadata in the previous inode into the new one. *Some* fields have to be set differently though!
4. Increment reference counts for the file blocks.
5. Insert an entry with the new inode and target name into the target directory. Most of the time you should be able to find space for your entry inside one of the existing allocated blocks. Sometimes however there won't be enough space for your entry; in which case you have to allocate a new data block and add it to the directory inode. You are expected to handle this case.

Your program should print the allocated inode number as well as the data block allocated for the directory if one has been allocated (-1 otherwise).

Example output where inode 17 was allocated and the directory entry fit inside an existing block:

```
17
-1
```

Another one where inode 126 was allocated and a new data block 1257 was allocated for the entry:

```
126
1257
```

The output alone is of course not enough, you should make sure to update the necessary filesystem structures correctly and that your filesystem remains consistent. The new file should be visible under the target directory, share blocks with the original file and you should not allocate any extra unnecessary blocks or corrupt other files. As a sharing test, you can try mounting the filesystem as ext2 and modifying one of the two files. Changes in one should be reflected in the other if the blocks have been shared.

Resulting images are not expected to match bitwise. Obviously, things like timestamps will be different. And there may be multiple valid positions for inserting an entry into a directory without allocating a new block; all of which will be accepted.

Important: Many implementations use smart heuristics when allocating blocks and inodes. You shall not. When allocating a new block or inode, simply go through each block group and their bitmaps in order and allocate the *first* free one you find. You must use this approach for your outputs to match.

4.2 rm - removing files (40 pts)

The next utility you will write is `rm`, which is pretty much the `rm` you're used to, except it's on ext2s. The command will be run with the following format:

```
$ ./ext2sutils rm FS_IMAGE DEST
```

`FS_IMAGE` and `DEST` are the same as with `dup`. `DEST` can be either a `dir_inode/target_name` pair or an `/absolute/path/to/target` and will again always be a regular file.

To remove a file entry from a directory, you need to do the following:

1. Find the inode of the target directory through traversal if given an absolute path. Otherwise it's already given.
2. Find the target entry in the directory data blocks. This will give you the inode of the target file.
3. You also have to remove the directory entry you've found. This is simple linked list removal: the previous entry will have to be made to point to the next one. For the first entry case, simply set the inode field to 0 to indicate that the entry should be skipped. If the entry was the only one on the data block, you should **not** deallocate the data block.
4. Time to move on to the inode. First of all, you should decrement the hard link count of the inode. If there are still hard links left, we can stop here. Whew! If there are no hard links left, we'll have to delete the inode. Uh oh...
5. Deallocate the inode from the bitmap.
6. Then, go through the blocks of the file and decrease their reference counts. If a block's reference count becomes zero after decrementing, also deallocate that block from the bitmap.

The program should once again output two lines. The first line will contain the inode of the target file. The next line will contain the block numbers of all *deallocated* blocks (those who've reached a reference count of zero and have been deallocated from bitmaps, not all blocks). If no blocks were deallocated, print -1. The order of the blocks does not matter; the line will be sorted during grading.

Some hypothetical runs for clarification:

```
# remove file useless.bin under directory inode 17
# the target inode turns out to be 25 and no blocks are deallocated
$ ./ext2sutils rm fs.img 17/useless.bin
25
-1
# remove file keys.txt under /secret/very_hidden/.sn34ky
# the target inode is 257 and seven blocks get deallocated
$ ./ext2sutils rm myfs.img /secret/very_hidden/.sn34ky/keys.txt
257
251 327 328 326 325 87 441
```

Similar considerations as before apply: Just the output is not enough! The filesystem should remain consistent, the entry should be removed from the directory, no extra blocks should be allocated or deallocated and other files should not be touched.

5 Bonus Opportunities

5.1 Indirect Block Support (20 pts)

For the base part of the homework, supporting files (including directories) only having direct blocks is enough. If your program can also handle files and directories containing indirect blocks (up to triple indirect), you will get bonus points.

Please note that indirect blocks are not different from data blocks in terms of being shared; they are also subject to being reference counted and will be duplicated rather than copied when using `dup`.

Under some circumstances, you may have to allocate multiple blocks when you need a new data block for a directory entry when performing `dup`. As an example, consider the case where all direct and single indirect blocks are full and double indirect has not been allocated yet. In this case, you will have to allocate two indirect blocks along with the data block for a total of 3 and you should print all three block numbers in the second line of the output. Their order does not matter, and which block gets assigned to which level does not matter either as long as the structure is correct.

Remember that files can have holes in ext2, which can raise some questions about how to allocate new data blocks for directories especially when indirect blocks are involved. Do not worry about that and assume that your directories will not have file holes.

5.2 share - sharing file blocks between separate files (20 pts)

The best part of having a file system in which we can share blocks is... Sharing blocks of course! Wouldn't it be great to be able to reduce disk usage by combining equal blocks of different files? Time to do it!

```
$ ./ext2sutils share FS_IMAGE FILES...
```

The `FILES...` argument represents one or more files, all of which will be given as absolute paths. The goal is going through all the data and indirect blocks in all the files, and sharing those who contain the same data. When multiple blocks containing the same data are found, the one with the smallest block number shall be kept and referenced by the others. All the other blocks found containing the same data shall have their reference counts decremented and possibly be deallocated. Since this is a command that's useful for large files, your implementation **should support indirect blocks** for this bonus.

Your output will contain one line for each shared block group. The first number in the line will contain the smallest block (that all the others will reference), and the rest of the line will contain pairs of the form `block_no:ref_count`; the block number of each block containing the same data and how many times they have been referenced. The order of the lines and the order of the pairs in each line does not matter. If no blocks containing shared data are discovered, the output will be empty.

As an example, consider the following run and scenario:

```
$ ./ext2sutils share e2.img /A /B /C /D
```

Let's say you find that block 3 in A, block 13 in B and block 12 in D contain the same data, all of which appear once. Then, you find that block 48 appearing four times in B and block 9 appearing once in C contain the same data. Finally, you also find that block 24 appearing three times in D and block 32 appearing once in D contain the same data. The following would be a valid output:

```
3 3:1 12:1 13:1
9 9:1 48:4
24 24:3 32:1
```

A non-sorted order is also valid:

```
9 9:1 48:4
24 32:1 24:3
3 13:1 3:1 12:1
```

In the end, references to block 12 and 13 will be replaced by references to block 3. The reference count of 3 will increase by two; 12 and 13's reference count will decrease by one and they will be deallocated if the counter reaches zero. References to block 48 will be replaced by references to block 9: block 9 will have four more references and 48 four less references. Something similar happens for block 24 and 32.

Please note that all blocks are considered together as a pool of blocks, we do not care about which block belongs to which file *at all*. Indirect blocks are also considered for sharing, just like data blocks. Once again, the filesystem has to remain consistent after the operation, and blocks should actually become shared, with removed blocks having their reference counts decreased and possibly being deallocated. Unrelated parts of the filesystem should not be corrupted during the operation.

As we will be working with large files, time complexity is important. A naive $O(n^2)$ double loop over the blocks is not going to cut it for large amounts of blocks! Use of proper data structures to achieve an $O(n)$ or $O(n \log n)$ complexity is critical. Example images and runs with time limits will be provided later.

6 Specifications

- Your code must be written in C or C++.
- Your implementation will be compiled and evaluated on the `ineks`, so you should make sure that your code works on them.
- Implementing only the `inode_no` and `dir_inode/target_name` formats will net you half points from `dup` and `rm`. You will get full points if you also implement absolute paths.
- You are free to use any standard libraries in your code.

- You are supposed to read the filesystem data structures into memory, modify them and write them back to the image. Mounting the filesystem in your code is forbidden; do not run any other executables from your code using things like `system()` or `exec*()`. This includes `robin` which is provided to help you test.
- 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.
- 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' homeworks are all considered plagiarism and strictly forbidden.
- Follow the course page on ODTUClass and COW for possible updates and clarifications.
- 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.

7 Tips and Tricks

- The given steps for implementing `dup` and `rm` are meant to guide you, but do not contain every single detail. As an example, the free block counts in the super block 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 filesystem image into memory is an excellent candidate for using `mmap`.
- 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 filesystem after modifying images to catch problems early. Backup your images.
- Do not just ignore all "multiply-claimed blocks" warnings from `e2fsck`; some may be legitimate programming errors on your part.

8 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
$ ./ext2sutils
```

If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points.

Late Submission: Remember that you had two free late days given to you at the beginning of the semester which you can use if you haven't yet. Otherwise, a penalty of $5 \cdot (\text{late days})^2$ will be applied to your final grade.

9 Appendix - More ext2s and robin Details

9.1 Consistency Checking

There are essentially three *types* of blocks that are valid in ext2s:

- **File blocks:** These blocks are the ones shared between files. They are referenced at least once and should be referenced as many times as they appear in file inodes.
- **Special blocks:** These are blocks like the superblock, block bitmap, inode table etc. They are not referenced from any inode since they are not part of a file, but are still allocated. Their reference counts should be one.
- **Free blocks:** These blocks are not referenced from any file nor allocated. They are free blocks, ready to be allocated. Their reference counts should be zero.

`robin check` will check all the blocks of the filesystem and report any block not matching one of the above three cases as *problematic*. Then it's up to you to figure out the bug.

9.2 The Special .block_refmap File

When an ext2 filesystem is converted into ext2s via `robin convert`, an extra inode is allocated and the refmap blocks are indexed as data blocks for this new file, which is added to the root directory as `".block_refmap"`.

This prevents `e2fsck` from complaining about having found allocated blocks that do not belong to any files. Also, being able to see the refmaps directly through a file is useful for debugging; you can mount the filesystem and then check or modify the refmaps through this file. All the refmaps appears contigously in the file, even though they are in different block groups on the filesystem. Note that the refmap in the last block group can have many unused entries, just like bitmaps. The refmaps will always take 32 blocks per block group.

Since your code will not be mounting the filesystem, it should modify the refmaps directly by getting their positions from the block group descriptors.