Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2024 - 2025

Homework 3 - File System Chronology in an EXT2 File System

Due date: 06 06 2025, Friday, 23:59

1 Introduction

In this assignment, you are tasked with implementing a program that will recreate the history of the file system for 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.

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.

- **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.

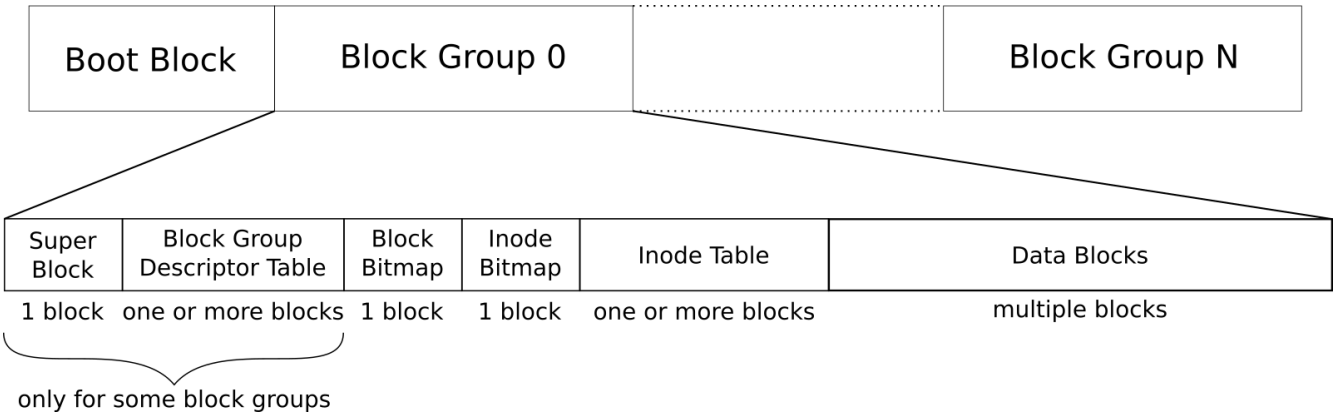

Figure 1: Overall ext2 Layout

  - **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.

- **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 change, 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

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 */
};
```

---

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]

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:

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.

[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!

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 */
};
```

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 change_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 */
};
```

- The BGD table is at the beginning of the first unoccupied block after the super block.
- 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.

Ext2 Directory Entry Structure

```c
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. */
};
```

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`

Important disclaimer: Some of the references mentions a creation time and not a change time. While the original ext2 documentation mentions a creation time, in order to be POSIX compliant, it is implemented as the change time in the Linux OS. A true change time was not introduced until ext4. More details of how timestamps works can be found in Section 3.2.

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 incks![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 incks 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 history recovery program histext2fs that will reveal the actions performed on the file system. Your program will also display the current state of the file system.

Actions that can be performed in the file system are limited to `touch`, `mkdir`, `rm`, `rmdir`, and `mv`. You can find more information about these actions on man pages. Your program needs to find the time the action, type of the action, arguments of the action, inodes that are directly effected by the action, and inodes of the directories the action took place in. You do not need to fill all of the fields to achieve full points, as that might be impossible.

To be able to this successfully, you have several guarantees.

1. No links will be used in the file system.
2. No file or directory name will contain space.
3. Deleted inodes will not be reused.
4. Contents of the deleted blocks will not be cleared and deleted blocks will not be reused.
5. Removed directory entries will not be deleted, instead, `length` of the `ext2_dir_entry` will be modified to cover the deleted entry.
6. Only one action per timestamp will be performed.
7. All actions will be performed from the root directory.
8. Minimum number of options will be used while actions are being taken to simplify the possibility space.
9. `rm` will not be used for directories.
10. `touch`, `mv`, and `mkdir` will always be called for an existing parent directory. Meaning there won't be any additional directory creation.
11. `touch` will only be used for non-existing files and it won't be used to update an existing file.
12. `mv` can be used to move directories.
13. Name of the entry can change when using `mv`.
14. `mv` won't be used in a way that the source and the destination are in the same directory.
15. Contents of the files won't be modified after creation.

More information about the specifics of the actions will be discussed in Section 3.2

3.1 Display the Current State (30 pts)

Your program will print whole the directory structure of the file system as a tree (without depth limit) with the inodes of the directory entries to a file which will be given to you. First part of this section only requires you to print existing directory entries, but in the second part you will also print directory entries that no longer exists.

3.1.1 Display the File Hierarchy (20 pts)

Your program will start printing 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:

```
- 2:root/
-- 11:lost+found/
-- 13:home/
--- 15:user1/
---- 22:file1.txt
---- 20:file2.txt
--- 21:user2/
---- 19:file3.txt
-- 12:etc/
--- 14: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). Also each directory entry is printed inode number first, name second, separated by the ":" character. 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 also a valid output:

```
- 2:root/
-- 11:lost+found/
-- 13:home/
--- 21:user2/
---- 19:file3.txt
--- 15:user1/
---- 22:file1.txt
---- 20:file2.txt
-- 12:etc/
--- 14:config.txt
```

While the following is not:

```
- 2:root/
-- 11:lost+found/
-- 13:home/
--- 15:user1/
---- 22:file1.txt
---- 20:file2.txt
---- 19:file3.txt
--- 21:user2/
-- 12:etc/
--- 14:config.txt
```

Be sure not to print the "./" or "../" directory entries, as you will lose points, and printing them might cause your code to get stuck in an infinite loop, which might cause you to lose all points.

3.1.2 Display the Ghost Entries (10 pts)

The gain additional points, you can also add ghost directory entries to your output. Ghost directory entries are directory entries which no longer resides in the directory, probably due to being moved all deleted. As previously in Section 3, these entries will be covered by their neighbors but will not get deleted or modified. While printing these ghost entries, you should enclose them in parenthesis, to signify that they are not actually there. If the ghost entry you print is a folder, do not print its contents. Aside from the presence of ghost entries, all other requirements from the previous section still apply. A possible output with ghost entries for the example given in the previous section can be as follows:

```
- 2:root/
-- 11:lost+found/
-- 13:home/
--- 15:user1/
---- 22:file1.txt
---- 20:file2.txt
---- (19:file1.txt)
--- 21:user2/
---- 19:file3.txt
---- (23:file4.txt)
--- (16:user3/)
-- 12:etc/
--- 14:config.txt
```

In addition to printing this, every information you gather during the printing process is useful in Section 3.2. In example, our previous output also tells us, there were previously "∼/home/user2/file4.txt" and "∼/home/user3/" directory entries that were deleted as well as "∼/home/user1/file1.txt" that was moved to " ∼/home/user2/" and renamed to "file3.txt".

3.2 Reveal the Action History (70 pts)

In this section, you will uncover the actions that was performed on the filse system by using a mixture of deleted but not wiped information and timestamps.

3.2.1 Actions

Here we will discuss all of the actions that can be performed on the test cases, their effects on the file system at large and their effect on the timestamps of the affected inodes in general. But before that, we need to discuss the timestamps in an inode and their purposes:

- `access_time` Updates when contents of an inode is specifically accessed. Because our test cases does not employ `ls`, `cd`, `cat` etc., it does not update and is a good substitute for creation time.
- `change_time` Updates when the metadata related to an inode changes. This include link count, but does not include `access_time`, due to it being considered non-essential metadata.
- `modification_time`: Updates when contents of the blocks that are being pointed by the inode changes.
- `deletion_time` Updates when the inode is deleted, either by `rm` or`rmdir` for our purposes.

With these in mind here is the specifics of each action:

1. **touch:** `touch PATH`

- **Purpose:** Creates an empty file with the path `PATH` or updates an existing file's timestamps (not relevant to us).
- **Timestamp:** Updates the `access_time`, `change_time`, and `modification_time` of the created inode, updates the `change_time` and `modification_time` of the parent directory's inode.

2. **mkdir:** `mkdir PATH`

   - **Purpose:** Creates an empty directory with the path `PATH`.
   - **Timestamp:** Updates the `access_time`, `change_time`, and `modification_time` of the created directory, updates the `change_time` and `modification_time` of the parent directory's inode.

3. **rm:** `rm PATH`

   - **Purpose:** Removes the file with the path `PATH`.
   - **Timestamp:** Updates the `change_time`, `modification_time`, and `deletion_time` of the deleted file's inode and updates the `change_time` and `modification_time` of the parent directory's inode.

4. **rmdir:** `rmdir PATH`

   - **Purpose:** Removes the directory with the path `PATH`. Only removes empty directories.
   - **Timestamp:** Updates the `change_time`, `modification_time`, and `deletion_time` of the deleted directory's inode and updates the `change_time` and `modification_time` of the parent directory's inode.

5. **mv (File):** `mv SRC DST`

   - **Purpose:** Changes the path of a file from `SRC` to `DST`. We will use a stricter version of `mv` compared to what exists in the Linux terminal. We will write the full path for the `DST`, not just the destination folder. When `SRC` and `DST` ends with different file names, this action also renames the directory entry. This can also be used to rename a file, by using the same parent path with different file names, but this usage (renaming without moving) won't be utilized in our test cases.
   - **Timestamp:** Updates the `change_time` of the moved file's inode and updates `change_time` and `modification_time` of the parent directory's inode.

6. **mv (Directory):** `mv SRC DST`

   - **Purpose:** Changes the path of a directory from `SRC` to `DST`. We will use a stricter version of `mv` compared to what exists in the Linux terminal. We will write the full path for the `DST`, not just the destination folder.
   - **Timestamp:** Updates the `change_time` of the moved directory's inode and updates `change_time` and `modification_time` of the parent directory's inode.

3.2.2 Output Format

You will output the action history into a file that will be give to you. Each line of the file must have the following format:

```
TIMESTAMP ACTION [ARGS] [AFFECTED_DIRECTORIES] [AFFECTED_INODES]
```

Square brackets are part of the output, enclosing the sections that might have multiple entries.

- `TIMESTAMP` is the timestamp of the action.
- `ACTION` is the type of the action, (ie. `rm`).
- `ARGS` are the arguments of the action, such as file paths.
- `AFFECTED_DIRECTORIES` are the inodes of the effected directories, meaning the directory in which effected directory entries reside.
- `AFFECTED_INODES` are the inodes of the effected directory entries.

For example, using the file system printed in Section 3.1.2, the move and renaming of the "∼/home/user1/file1.txt" might be described by the following output:

```
1746630723 mv [/home/user1/file1.txt /home/user2/file3.txt] [15 21] [19]
```

Order of arguments matters in this context, as the first one describes the source and the second one describes the destination. But the order of inodes are not important.

You might have missing information, that might cause you to not have the full picture. In that case, you can substitute any parts of the output with '?'. For example, lets assume the `modification_time` of the directory "∼/home/user2/" is before the deletion time of the inode 23, meaning the directory entry "∼/home/user2/file4.txt" is moved prior to its deletion. With no additional information, we can not say to where and when, meaning we might need to output

```
? mv [/home/user2/file4.txt ?] [21 ?] [23]
1746630745 rm [?] [?] [23]
```

Also, if `modification_time` both inode 15 and inode 21 is identical, we can not know when the move happened, meaning our output might need to be

```
? mv [/home/user1/file1.txt /home/user2/file3.txt] [15 21] [19]
1746630723 ? [?] [21] [?]
```

There might not be a way of knowing whether this both actions refers to the same event or not. Due to this, your output will be compared with the ground proof as follows:

1. All of the output lines with existing timestamps will be processed and will get associated with an action in the ground truth.
2. Remaining output lines are processed in order, and gets associated with the first action in the ground truth which:
   - it describes, meaning all of the given fields of the output matches with action in the ground truth.
   - it provides additional info about, meaning it completes at least one field that is not being completed by any of the outputs that are already associated with it.

If an output does not fulfill these two conditions for any action in the ground truth, you will lose points.

In addition to this, each output line must have at least two fields filled, otherwise, it will be ignored. If the `ACTION` is the part of the output that is missing, number of fields in the possibly multi-field sections does not matter. For example, in the previous output example,

```
? mv [/home/user1/file1.txt /home/user2/file3.txt] [15 21] [19]
1746630723 ? [?] [21] [?]
```

If both outputs refers the same action, it can be argued that the correct second line should be `1746630723 ? [?] [21 ?] [?]`, but because we do not know that, you can use either.

3.3 Grading

First, for the purpose of displaying current states of the file system, you will be graded by the percentage of how many entries you printed. There is no partial point for directory entries with wrong inode numbers or inode numbers with wrong names, both counts as an erroneous output. For an erroneous output, you will lose the point from one of your correct outputs. You can not get negative points due to this. In addition to this, regular and ghost entries are graded separately, meaning a penalty from one can not reduce your grade coming from the other one.

Secondly, grades for the "Reveal the Action History" part will be decided by comparing the number of fields the number of fields that are theoretically possible to recover. The theoretical maximum will be counted as 120% of recovery and you will be graded according to how many fields you recovered. This 120% scaling for the theoretical maximum exists for you to make sure you can get 100% points without having access to all test cases that will be used during grading and it is not extra points. Also for some cases, all info can be recoverable, but this scaling gives you some amount of leeway.

Finally, your code needs to end its execution within 2 minutes for each test case. If it takes more than that, you will lose 1% of your grade for that test case per second.

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. 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*().
4. Including POSIX ext2/ext3/ext4 libraries (as well as kernel codes and their variations etc.) is not allowed.
5. 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.
6. Your submission should include every file you used in your code. No files will be supplied to you.
7. 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.
8. Follow the course page on ODTUClass for possible updates and clarifications.
9. 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

- Printing the current state of the file system, especially when printing the ghost entries, uses many functionalities that is also necessary when revealing the history of the file system and is a lot simpler in logic. Due to this, I recommend starting there.
- It is trivial to find deletion time of all inodes, meaning in example cases that uses `rm` and `rmdir`, you have some guaranteed points.

- Don't forget that there is only one action per `TIMESTAMP` value, if `modification_time` of two folders are the same, last action that happened to them affected both of them.
- Because entries without `TIMESTAMP` fields processed in the order they are printed, it is a good idea to try to print actions in chronological order.
- To prevent losing points due to grader matching your output with a different action, be as precise with your outputs as possible. This is an intended behavior, not a bug with the grader.

6 Submission

Submission will be done via ODTUClass. Create a gripped 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
$ ./histext2fs image_location state_output history_output
```

If there is a mistake in any of these steps mentioned above, you will lose 10 points.
Late Submission: There is no late submissions for this assignment.