DiskEdit

Quentin Fringhian, Pierre Aillet

December 16, 2022

Contents

1 Introduction

Welcome to **DiskEdit**, a program that allows you to edit a disk image with a simple file system. It provides a text-based user interface (TUI) that helps you create files, directories, and edit existing files.

This document is a report of the project 3, which was developed as part of the "Operating Systems" course at Dankook University. In this report, we will explain how to use the program, how it works, and how to compile it. We will focus on the features that allow you to edit the disk image and will not discuss the implementation of the TUI in detail.

If you have any questions or would like to learn more about the TUI, please send us an email.

Purpose The purpose of DiskEdit is to provide a simple and intuitive tool for editing disk images. It is designed to be easy to use, even for users who are new to operating systems or disk image editing. With DiskEdit, you can create and modify files and directories on a disk image, just as you would on a regular computer.

Features Some of the main features of DiskEdit include:

- Creating and deleting files and directories
- Editing the contents of files
- Displaying the directory structure and file metadata

2 Running the Program

To run the DiskEdit program, you will need to compile it first. To do this, navigate to the root directory of the project and run the following command:

```
make
```

This will create an executable file called ‘diskEdit‘ in the root directory of the project. You can then run the program by using the following command:

```
./diskEdit <disk image>
```

where ‘<disk image>‘ is the path to the disk image that you want to edit.

Text-based User Interface (TUI) When you run the program, it will open the TUI, which allows you to navigate through and edit the contents of the disk image. The TUI consists of two main parts: the directory view and the file view.

2.1 Directory View

In the directory view, you can use the UP and DOWN arrow keys to move between different files and directories. To enter a directory, you can press the ENTER key on the selected directory.

The TUI also provides a command prompt that you can use to perform various actions. To access the command prompt, you can press the ':' key. This will open a prompt where you can enter commands.

Here is a list of the commands that you can use in the directory view:

- `quit` or `q`: Quit the program
- `touch`: Create a new file. The TUI will prompt you to enter the name of the file you want to create.
- `mkdir`: Create a new directory. The TUI will prompt you to enter the name of the directory you want to create.
- `rm`: Delete the currently selected file.
- `rmdir`: Delete the currently selected directory, including all of its contents (files and subdirectories).

2.2 File View

To view or edit the contents of a file, you can press the ENTER key on the selected file in the directory view. This will open the file view, which allows you to navigate through and edit the contents of the file.

In the file view, you can use the UP and DOWN arrow keys to move between lines in the file, and the LEFT and RIGHT arrow keys to move within a line.

To enter a command in the file view, you can press the ':' key to access the command prompt. From the command prompt, you can enter the following commands:

- `edit` or `e`: Enter edit mode, which allows you to modify the contents of the file. To exit edit mode, you can press the ESC key.
- `save` or `s`: Save your changes to the file.

3 Explanation of How the Program Works

In this section, we will delve into the details of how DiskEdit interacts with the disk image. First, we will examine the file system used by the disk image and the various structures that make up this file system to have a better understanding of how DiskEdit works. Then, we will see how DiskEdit uses these structures to perform various operations on the disk image.

3.1 File System

The disk image that DiskEdit operates on is divided into three main parts: the super block, the inode table, and the data blocks. These three parts work together to store and manage the files and directories on the disk image.

Super Block The super block is a special block that contains important information about the overall structure and layout of the file system. It is usually located at the beginning of the disk image and contains the following fields:

- `partition_type`: Indicates the type of partition used by the file system (e.g., ext2, fat32, etc.).
- `block_size`: The size of a block in bytes.
- `inode_size`: The size of an inode in bytes.
- `first_inode`: The index of the root inode in the inode table.
- `num_inodes`: The total number of inodes in the file system.
- `num_inode_blocks`: The number of blocks reserved for the inode table.
- `num_free_inodes`: The number of free inodes available for use.
- `num_blocks`: The total number of blocks in the file system.
- `num_free_blocks`: The number of free blocks available for use.
- `first_data_block`: The index of the first data block.
- `volume_name`: A string containing the name of the volume.
- `block_bitmap`: A bitmap that indicates which blocks are free and which are used.
- `padding`: Padding to ensure that the super block is a fixed size.

DiskEdit primarily uses the `block_size`, `inode_size`, `first_inode`, and `block_bitmap` fields from the super block.

Inode Table The inode table is a contiguous block or blocks that contain information about the files and directories on the disk image. Each inode corresponds to a single file or directory and contains the following fields:

- `mode`: Indicates the type of the file or directory (e.g., regular file, directory, device file) and the permissions associated with it.
- `locked`: A flag that indicates whether the file is currently being written to.

- `date`: The date and time when the file or directory was created or last modified.
- `size`: The size of the file in bytes.
- `indirect_block`: A block number that points to an indirect block, which in turn points to the data blocks containing the contents of the file. If this field is set to -1, it indicates that the file does not use an indirect block.
- `blocks`: An array of block numbers that directly point to the data blocks containing the contents of the file.

DiskEdit primarily uses the `mode`, `size`, and `blocks` fields from the inode.

Data Blocks Data blocks are the actual blocks that store the contents of the files and directories on the disk image. They are simply arrays of bytes with the following structure:

```
struct blocks {
    unsigned char d[1024];
};
```

The inodes and the data blocks work together to store the contents of the files and directories on the disk image. The inodes contain metadata about the files and directories, such as their sizes and permissions, as well as pointers to the data blocks that contain their contents. The data blocks, on the other hand, contain the raw data of the files and directories.

With this understanding of the file system used by the disk image, we can now see how DiskEdit manipulates and manages the files and directories on the disk image.

3.2 DiskEdit Commands

DiskEdit provides a number of commands that allow you to perform various operations on the disk image, such as creating, deleting, and modifying inodes and blocks.

3.2.1 Loading the Disk Image

Before you can use the various DiskEdit commands, you must first load the disk image into the program. When you load a disk image, DiskEdit reads the necessary information from the disk image file and stores it in a special data structure called the `disk` structure. This structure contains the following fields:

- `super_block`: The super block of the disk image, which contains information about the overall structure and layout of the file system.

- `inode_table`: The inode table of the disk image, which contains information about the files and directories on the disk image.
- `current_dir`: The inodes index of the files and directories in the currently loaded directory.
- `cache`: A cache that stores the contents of the files and directories that have been loaded into memory.
- `blocks_offset`: The offset of the data blocks in the disk image file.
- `d_path`: The path of the disk image file.

To optimize memory usage, DiskEdit only loads the necessary information from the disk image file into memory and does not read the data blocks until they are actually needed. This allows DiskEdit to efficiently handle large disk images without using excessive memory.

When you load a disk image, the program will automatically load the root directory into memory. You can then use the various DiskEdit commands to navigate and manipulate the files and directories on the disk image.

3.2.2 Manipulating Blocks

This sub-subsection covers the various operations that can be performed on blocks in the disk image, including reading, allocating, deleting, and editing blocks.

Reading a Block

The `read_block` command is used to retrieve the contents of a block in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `block`, indicating the index of the block to be read.
- An unsigned integer, `size`, indicating the size of the block to be read.

The command returns a `block_buff_t` structure, which consists of the following fields:

```
1 typedef struct block_buff_s {
2     int block_number;
3     unsigned char *data;
4 } block_buff_t;
```

- The data of the block is stored in the `data` field of the structure.
- The `block_number` field contains the index of the block that was read.

When this command is executed, it first checks the cache for the requested block. If the block is present in the cache, it is returned from there. Otherwise, the block is read from the disk image file, stored in the cache, and then returned to the caller. For further details, refer to the `read_block` function in the `block_load.c` file.

Allocating a Block
The `alloc_block` command is used to allocate a new block in the disk image. It takes the following argument as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.

The command returns an integer, which indicates the index of the newly allocated block, if successful. Otherwise, it returns -1.

When this command is executed, it searches the block bitmap in the super block of the disk image for an available block. As soon as it finds a free block, it updates the corresponding bit in the bitmap to 1 and returns the index of the newly allocated block. For further details, refer to the `alloc_block` function in the `block_create.c` file.

Deleting a Block
The `free_block` command is used to free a specified block in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `block`, indicating the index of the block to be freed.

The command does not return any value.

When this command is executed, it updates the corresponding bit in the block bitmap to 0. For further details, refer to the `free_block` function in the `block_create.c` file.

Editing a Block
The `block_edit` command is used to edit the contents of a block in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `block`, indicating the index of the block to be edited.
- An unsigned char pointer, `data`, which points to the data to be written to the block.

- An unsigned integer, `size`, indicating the size of the data to be written to the block.

The command does not return any value.

When this command is executed, it writes the data pointed to by the `data` argument to the specified block in the disk image. For further details, refer to the `edit_block` function in the `block_edit.c` file.

3.2.3 Manipulating Inodes

This sub-subsection covers the various operations that can be performed on inodes in the disk image, including reading, allocating, deleting, and editing inodes.

Reading an Inode

The `read_inode` command is used to retrieve the contents of an inode in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `inode`, indicating the index of the inode to be read.

The command returns a `inode_buff_t` structure, which consists of the following fields:

```
1 typedef struct inode_buff_s {
2     int inode_number;
3     unsigned short *blocks;
4     unsigned char *data;
5 } inode_buff_t;
```

- The data of the inode is stored in the `data` field of the structure.
- The `inode_number` field contains the index of the inode that was read.
- The `blocks` field contains the indices of the blocks that are allocated to the inode.

When this command is executed, it will retrieves an inode from the inode table in the super block of the disk image. It then copies the indexes of the blocks allocated to the inode into the `blocks` field of the `inode_buff_t` structure. Finally, it reads each allocated block using the `read_block` command until it has read the equivalent of the size of the inode. For more information, see the `read_inode` function in the `inode_load.c` file.

Allocating an Inode
The `alloc_inode` command is used to allocate a new inode in the disk image. It takes the following argument as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An unsigned int, `mode`, indicating the mode of the inode to be allocated.

The command returns an integer, which indicates the index of the newly allocated inode.

When this command is executed, it searches the inode table in the super block of the disk image for an available inode. Upon finding a free inode, it updates the inode table and sets the mode of the inode to the value specified by the `mode` argument, the size of the inode to 0, the date of the inode to the current date, and the list of blocks allocated to the inode to 0. For more information, see the `alloc_inode` function in the `inode_create.c` file.

Deleting an Inode
The `free_inode` command is used to free a specified inode in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `inode`, indicating the index of the inode to be freed.

The command does not return any value.

When this command is executed, it frees all the blocks allocated to the inode using the `free_block` command. It then updates the inode table to indicate that the inode is free. For more information, see the `free_inode` function in the `inode_create.c` file.

Editing an Inode
The `inode_edit` command is used to edit the contents of an inode in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `inode`, indicating the index of the inode to be edited.
- An unsigned char pointer, `data`, which points to the data to be written to the inode.
- An unsigned integer, `size`, indicating the size of the data to be written to the inode.

The `inode_edit` function returns an integer indicating the success or failure of the operation. If the operation is successful, it returns 0. If the file is not editable, it returns -1. If the file is locked, it returns -2. If a block could not be allocated, it returns -3.

When this command is executed, it first checks if the inode is editable. Then, block by block, it rewrites the data of the already allocated blocks using the `edit_block` command. If there are no more allocated blocks available, it allocates a new block using the `alloc_block` command. If the data to be written is smaller than the size of the inode, it frees the remaining blocks using the `free_block` command. Finally, it updates the size of the inode to the size of the data written. For more information, see the `inode_edit` function in the `inode_edit.c` file.

3.2.4 Manipulating Directories

This sub-subsection covers the various operations that can be performed on directories in the disk image, including reading directories, changing directories, saving directories, creating files and directories, and deleting files and directories.

Reading a Directory

The `read_dir` command is used to retrieve the contents of a directory in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `inode`, indicating the index of the directory to be read.

The command returns a `dir_buff_t` structure, which consists of the following fields:

```
1 typedef struct dir_buff_s {
2     int dir_inode_number;
3     file_list_t *file_list;
4 } dir_buff_t;
```

- The `dir_inode_number` field contains the index of the directory that was read.
- The `file_list` field contains a list of the file information of the files and directories in the directory.

The file information is stored in the following structure:

```
1 typedef struct file_s {
2     unsigned char *name;
3     int inode_number;
4 } file_t;
```

- The `name` field contains the name of the file.
- The `inode_number` field contains the index of the inode of the file.

When this command is executed, it first check if the inode is a directory. If it is not, it returns a `NULL` pointer. Otherwise, it reads the inode of the directory using the `read_inode` command, the data retrieved from the inode is then converted, it contains the list of inodes of the files and directories in the directory. For more information, see the `read_dir` function in the `dir_load.c` file.

Change of Directory
The `dir_change` command is used to change the current directory in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- An integer, `inode`, indicating the index of the directory to be changed to.

The command doesn't return any value.

When this command is executed, it first checks if the inode is a directory. If it is not, it doesn't change anything. Otherwise, it reads the directory using the `read_dir` command and updates the current directory in the disk image to the directory read. For more information, see the `dir_change` function in the `dir_change.c` file.

Saving a Directory
The `save_dir` command is used to save the contents of a directory in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- A file_list_t pointer, `file_list`, which points to the list of files and directories in the directory.
- An integer, `inode`, indicating the index of the directory to be saved.

The command doesn't return any value.

When this command is executed, it first change the list into an array unsigned char, and then writes the array to the inode using the `inode_edit` command. For more information, see the `dir_save` function in the `dir_edit.c` file.

Creating a File
The `touch` command is used to create a file in the disk image and add it to the current directory. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- A string, `file_name`, which contains the name of the file to be created.

The command returns an integer indicating the inode index of the file created. If the file could not be created, it returns -1.

When this command is executed, it first alloc a new inode using the `alloc_inode` command, indicating that it is a file in the mode argument. Then it add the file to the list of files in the current directory. Finally, it saves the current directory using the `save_dir` command. For more information, see the `touch` function in the `touch.c` file.

Creating a Directory
The `mkdir` command is used to create a directory in the disk image and add it to the current directory. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- A string, `dir_name`, which contains the name of the directory to be created.

The command returns an integer indicating the inode index of the directory created. If the directory could not be created, it returns -1.

When this command is executed, it first alloc a new inode using the `alloc_inode` command, indicating that it is a directory in the mode argument. It will add the `.` and `..` files to the directory. Then it add the directory to the list of files in the current directory. Finally, it saves the directory and the current directory using the `save_dir` command. For more information, see the `mkdir` function in the `mkdir.c` file.

Removing a File
The `rm` command is used to remove a file in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- A dir_buff_t pointer, `dir_buff`, which points to the structure containing information about the directory containing the file.
- An integer, `inode`, indicating the index of the file to be removed.

The command doesn't return any value.

When this command is executed, it will look for the file in the directory and remove it from the list of files in the directory. Then it will free the inode of the file using the `free_inode` command. Finally, it will save the directory using the `save_dir` command. For more information, see the `rm` function in the `rm.c` file.

Removing a Directory

The `rmdir` command is used to remove a directory and it's content in the disk image. It takes the following arguments as input:

- A `disk_t` pointer, `disk`, which points to the structure containing information about the disk image.
- A dir_buff_t pointer, `dir_buff`, which points to the structure containing information about the directory containing the file.
- An integer, `inode`, indicating the index of the file to be removed.

The command doesn't return any value.

When this command is executed, it will read the content of the directory using the `read_dir` command. Then it will check all the files and directories in the directory. If the file is a directory, it will call the `rmdir` command recursively. If the file is a file, it will call the `rm` command (It will nevertheless not remove the `.` and `..` files). Finally, it will remove the directory from the list of files in the parent directory and free the inode of the directory using the `free_inode` command and save the parent directory using the `save_dir` command. For more information, see the `rmdir` function in the `rmdir.c` file.

4 Conclusion

This project gave us the opportunity to learn more about the Linux file system . While we were unable to include all of the features we had intended due to time constraints, we are open to any suggestions or questions you may have on how to further improve the project. Please don't hesitate to reach out to us if you have any feedback or inquiries. We hope you have a wonderful day.