# ASSIGNMENT TWO

*[COMP3300]*

*Boyu Dai, Haoting Chen, Yichi Zhang*

*[ANU] | [School of Computing]*

## Table of Contents

## 1 Introduction

The goal of this project is to implement a simple Linux file system “vvsfs” that supports some common file operations, such as creating, deleting, and renaming files and directories. The artefact of the project is a Loadable Kernel Module “vvsfs.ko” that can be loaded and unloaded onto Linux. The file system can then be used to format a virtual or physical block device so that files can be transferred in and out or manipulated on this device.

The Linux kernel provides a uniform interface, the Virtual File System (VSF), that can group various file systems together so that files can be shared in between. The purpose of a file system is to connect the VSF layer and the block device layer. It defines how files are stored in the data block of a block device, which can sometimes affect the performance and security of file manipulation in the user space. Data blocks of a block device can be further translated into SSD blocks or HDD sectors through device drivers to store information persistently.

## 2 Linux Kernel Background

"Linux Operating System" are a selection of operating systems based on the Linux Kernel[1]. The kernel is the fundamental building block of the daily-used operating system. It is responsible for communicating with hardware, handling system calls and performing context switching. However, the size of kernel source code is usually huge. When developing kernel codes, a lot of time is spent on compiling the kernel. Meanwhile, kernels are generally not designed to be hot plugging and unplugging. The user has to reboot to install and run a new kernel version.

Loadable Kernel Module (LKM) is an alternative for adding new functions to the kernel without recompiling the whole kernel. LKM is a part of the kernel code that is not bound to the image from which the Operating system is booted. Developing, Debugging and Maintaining LKM is much faster and easier than the base kernel. LKM could also be loaded and unloaded when users are not using it. These advantages make LKM a competitive choice for developing file system drivers.

In this project, we implemented a file system driver as an LKM. A file system driver specifies a file system's on-disk data structures and access methods. The driver handles system calls related to reading, writing and searching files on the supported file system. It is also responsible for maintaining data structures that might be used for various purposes like journaling.

## 3 Structure and Specification

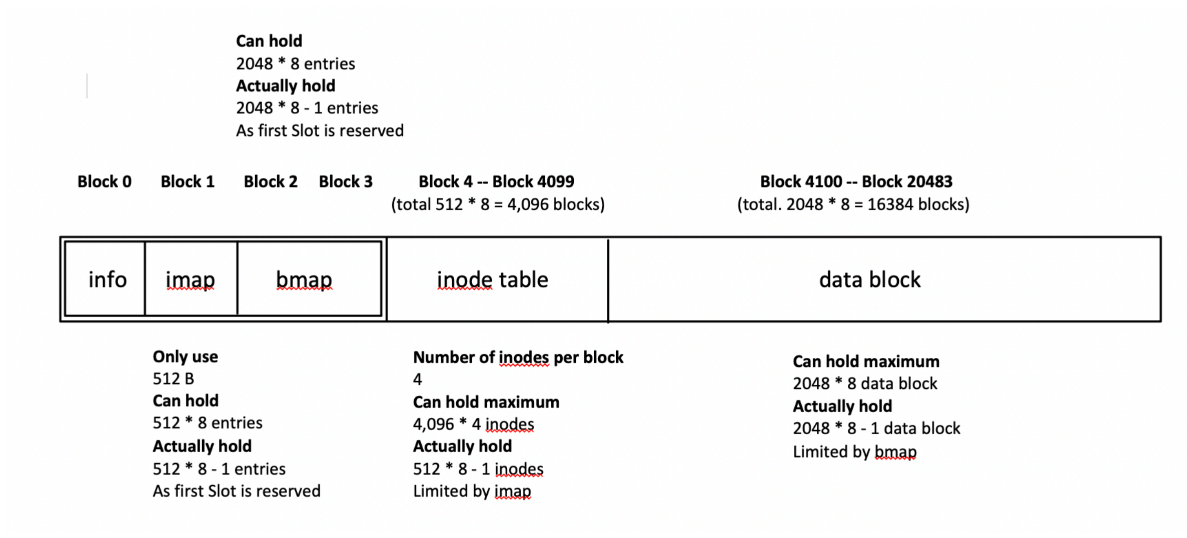
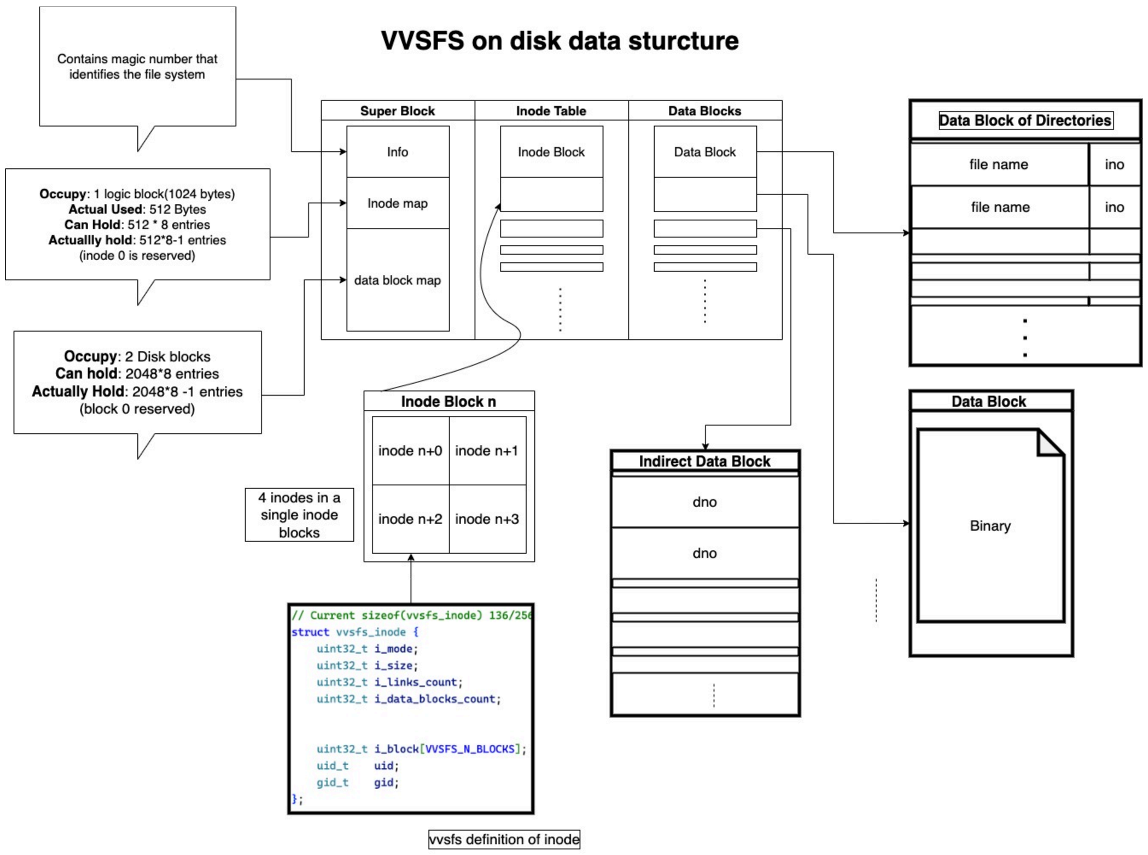


Figure 1: blocks in VVSFS

[1] J. Eckert, Linux+ Guide to Linux Certification. Cengage Learning, 2012.



```
// Current sizeof(vvsfs_inode) 136/256
struct vvsfs_inode {
    uint32_t i_mode;
    uint32_t i_size;
    uint32_t i_links_count;
    uint32_t i_data_blocks_count;


    uint32_t i_block[VVSFS_N_BLOCKS];
    uid_t    uid;
    gid_t    gid;
};
```

Figure 2: details of blocks in VVSFS

The file system we implemented has a fixed size of 20484 blocks, with each block equal to 1024 B. As shown in Figures 1 and 2, VVSFS manages three collections of data blocks.

**Super Block** (Block 0 -3) contains info block that stores the file system information, inodes bitmap (4096 entries) table and a data block bitmap (16384 entries) table that manages the usage.

**Inode Table** (Block 4 - 4099) uses 4096 disk blocks (1024 B), since each inode size is 256 B, it can store up to (4096 * 4 inode). However, only ¼ is usable since inode bitmap only has 4096 entries. The inode contains meta of a file, shown in Figure 2. Most importantly, it has 15 pointers pointing to a data block. For a file, 14 of them are direct and 1 is indirect, which further pointer to a data block containing 256 pointers.

**Block Region** (Block 4100 – 20483) keeps the actual content of files in the system. It can store,

-  Up to 1024 bytes of raw data of a file.
-  Up to 256 pointers pointing to other data blocks.
-  Up to 8 directory entry mapping between file name and inode number.

Tables 1 and 2 provide some further information.

| | |
|---|---|
| Max number of inodes (files) excluding root dir | 512 * 8 – 1 = 4095 files |
| Num of direct pointers for file | 14 pointers |
| Num of indirect pointers for file | block size (1024) / pointer size (4) = 256 pointers |
| Total Num of data block pointers for file | 14 + 256 = 270 pointers |
| Max file size | 270 kB |
| | |
| Total Num of data block pointers for directory | 14 pointers |
| Max Num of directory entries | 14 * (1024 - 128) = 112 entries |

Table 1: the maximum file and directory size

| **Operation name** | **Description** |
|---|---|
| create | Create an inode |
| lookup | Look up a dentry in a directory |
| mkdir | Create a directory |
| unlink | Delete a file |
| rmdir | Delete a directory |
| rename | Rename a file or a directory |
| llseek | Seek to a position in a file |
| fsync | Synchronize a file's in-memory state with the storage |
| read_iter | Read data from a file |
| write_iter | Wead data from a file |

Table 2: Operation Specification

## 3 High-Level Description of File System Operations

As shown in Figure 3, the operation of a file system is highly related to VFS, buffer headers, and disk. VVSFS use buffer header to get the cache of a block on the disk. Manipulate in the memory space and then write back to the disk when it is changed. When VFS ask for information about the file system, such as inode and dentry, the result sent by VVSFS will also be cache in the dcache in VFS, so that other kernel modules can easily access the disk through VFS. The following are the description of the main operations.

Create:

- A user program issues a request to create a new file.
- VFS receives the request and forwards it VSSFS.
- VSSFS look for available blocks through inode and data block bitmap.
- 1 inode block and 1 and more data block is allocated.
- The pointers in the inode block are pointed to the allocated data block.
- Notify VFS to update its dcache as well as sync change into the disk.

unlink

- A request to delete a file is made by user.
- VFS invokes the unlink operation on vvsfs.
- VVSFS marks the inode and its data block as deleted and removes the dentry from its parent directory.
- Update the block for the parent directory by writing to the cache and eventually flush to the disk.
- * rmdir is similar to unlink with the addition of checking empty directory at the beginning

lookup

- In user space, a program requests access to a file or directory by its name.
- VFS provides VVSFS the parent folder's inode, and dentry containing the file name.
- VVSFS cache the blocks of the file folder from the disk and looks at its dentry table to find an inode associated with the file name.
- Return the retrieved inode back to VFS.
-

Rename

- At user space: The user calls the system call to rename the file.
- VFS processes the syscall and collects the necessary information (inode, dentry …) for the renaming operation
- Relevant functions in VVSFS are called with the necessary information.
- VVSFS modifies relevant data structures on the disk and Dcache (writing may be pended)
- The process switches back to the user space

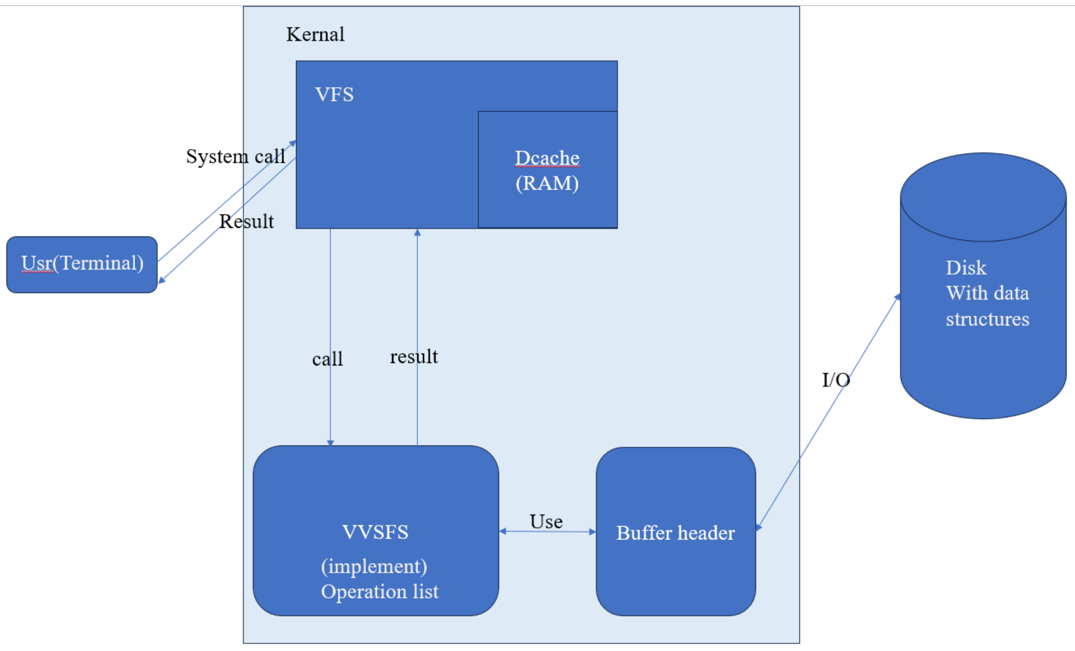


Figure 3: relation between VVSFS, VFS, butter header, disk, and user space

## 4 Implementation Detail of File System Operations

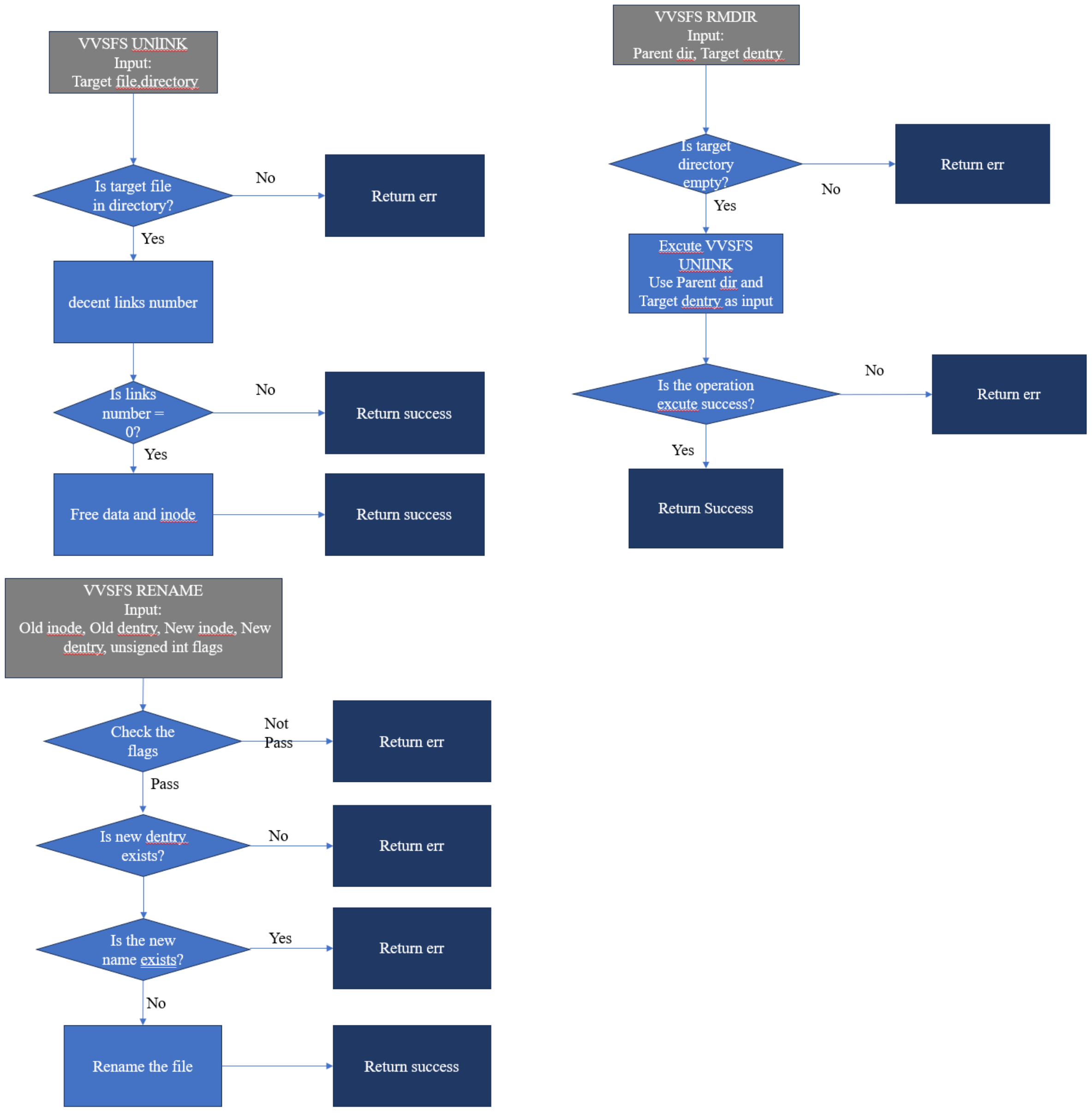


Figure 4: The flow chart of three inode operations

Figure 4 are the implementation of the three operations of inode. The general idea is as follows:

vvsfs_unlink:

Input: directory filename

Initially, check if the file exists in the directory. If the corresponding filename is not found within the input directory, return an error indicating "File not found."

If the file is indeed present in the directory, decrease the number of hard links associated with the current file.

If the number of links for the current file is reduced to 0, release the data storage space of the current file and the inode space. Return success in this case.

If the number of links for the current file is not 0, it implies that the file still has other links, so return a reduction success status.

vvsfs_rmdir:

Input: The name of the parent directory. The dentry of the directory to be deleted

Initially, verify if the directory is empty. If the directory contains any files, return an error.

If the directory is confirmed to be empty, execute the vvsfs_unlink function using the parent directory and the target directory as input.

The returned value from this operation depends on the outcome of vvsfs_unlink. If successful, return a success status; otherwise, return an appropriate error.

vvsfs_rename:

Input: old inode, old dentry, new inode, new dentry, status value flag

Begin by checking the provided flag. If the flag is not set to RENAME_EXCHANGE or RENAME_NOREPLACE, return an error.

Verify the validity of the new dentry. If the new dentry is found to be invalid, return an error.

Check whether a file with the same name exists in the directory. If a file with an identical name is found, return an error indicating the presence of a conflicting file.

If no conflicting file is found, proceed with the renaming operation, and return a success status.

## 5 Key Mechanisms of Implement

### 5.1 vvsfs_find_entry

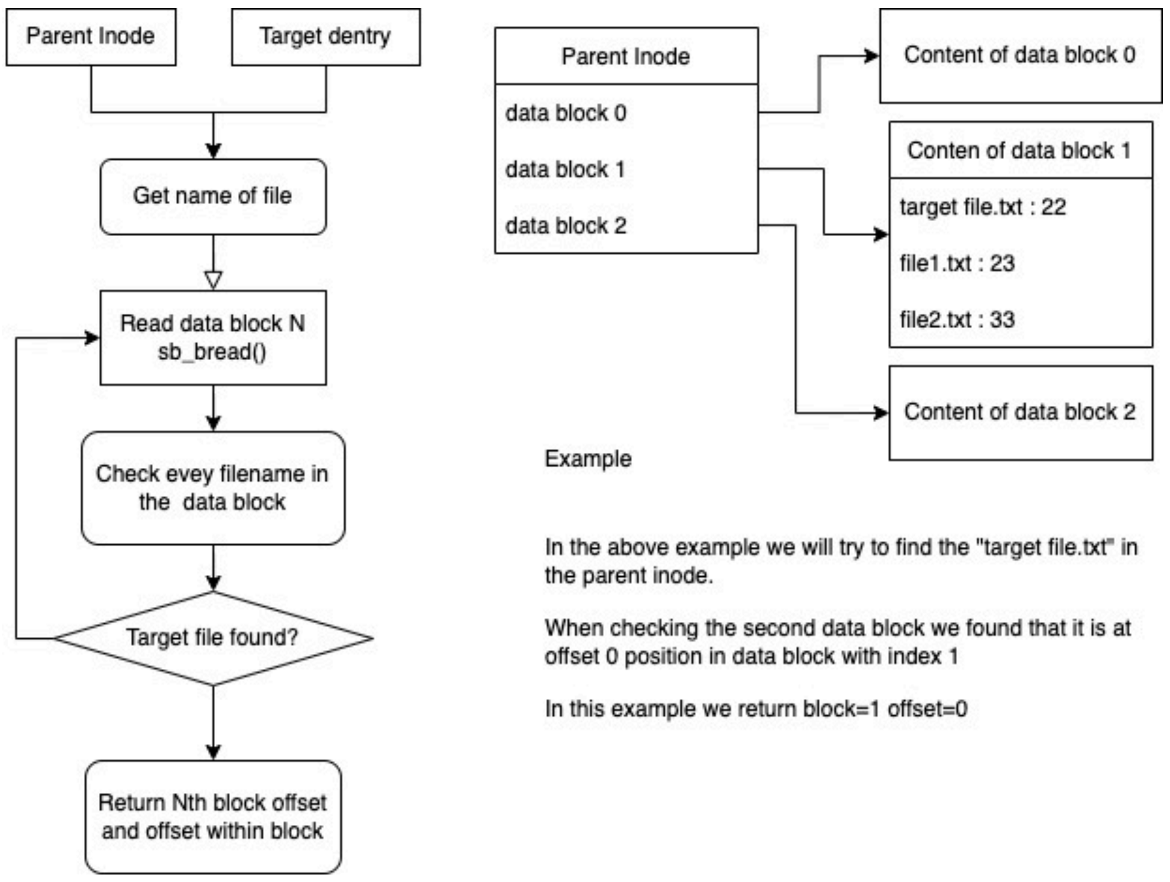

Parent Inode
Target dentry
Get name of file
Read data block N sb_bread()
Check evey filename in the data block
Target file found?
Return Nth block offset and offset within block
Parent Inode
data block 0
data block 1
data block 2
Content of data block 0
Conten of data block 1
target file.txt : 22
file1.txt : 23
file2.txt : 33
Content of data block 2
Example
In the above example we will try to find the "target file.txt" in the parent inode.
When checking the second data block we found that it is at offset 0 position in data block with index 1
In this example we return block=1 offset=0

Figure 5: mechanism of finding entries in a directory.

The vvsfs file system allows users to create, rename and delete files in the file system. These operations are possible because vvsfs implements procedures/methods specified by the Linux Virtual File system. For deleting files, developers must implement the unlink function and put the implementation in the corresponding position in inode operations.

In our implementation of unlink, we copy the last entry (file and ino pair) in the last data block to the position where the entry we want to remove resides. Next, we clear the memory of the last entry in the last data block and update the meta info.

In the unlink process, we need to find the entry for the target block, as shown in Figure 5, before we can perform any further operations on it. One of the crucial functions in our implementation is the "vvsfs_find_entry".

## 5.2 indirect pointers block

There is a very important data structure here, which is indirect pointers. Figure 6 gives a good example:

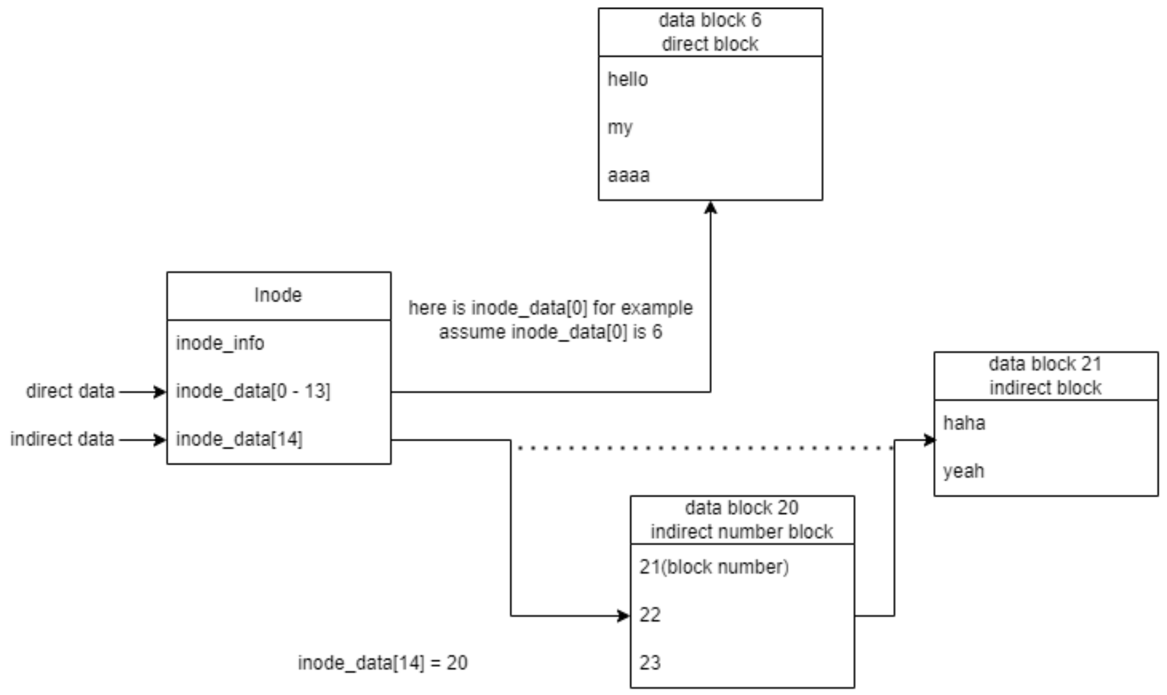


Figure 6: an example of indirect pointers

First of all, there are pointers to a total of 15 data blocks from 0 to 14 in the inode. Among them, the pointer with an index value of 0 to 13 is a direct index. For example, inode_data[0], its value is 6, which means it points to data block 6. , and data block 6 directly stores the file data. But once the file size exceeds the size of 14 data blocks, an indirect pointer with an index value of 14 must be used. inode_data[14] points to a data block that stores pointers, and the pointer in this data block points to: data block 21 is where the file data is actually stored.

## 5.3 How to free a file involving indirect blocks (vvsfs_unlink)

As shown in Figure 7, to initiate the process, the first step is to determine the existence of an indirect pointer. If there is no indirect pointer present, the procedure involves direct data block deletion based on the direct pointers. Subsequently, the inode data is deleted.

In case an indirect pointer is found, the process is slightly more intricate. Start by accessing the last pointer to retrieve the address of the indirect pointer data block.

Next, traverse through the data blocks pointed to by the pointers within the indirect pointer data block. For each of these data blocks, delete the data contained within.

After successfully cleaning up the data within the data blocks pointed to by the indirect pointers, return to delete any remaining data based on the direct pointers.

Finally, once all data has been properly handled, proceed to delete the inode data.

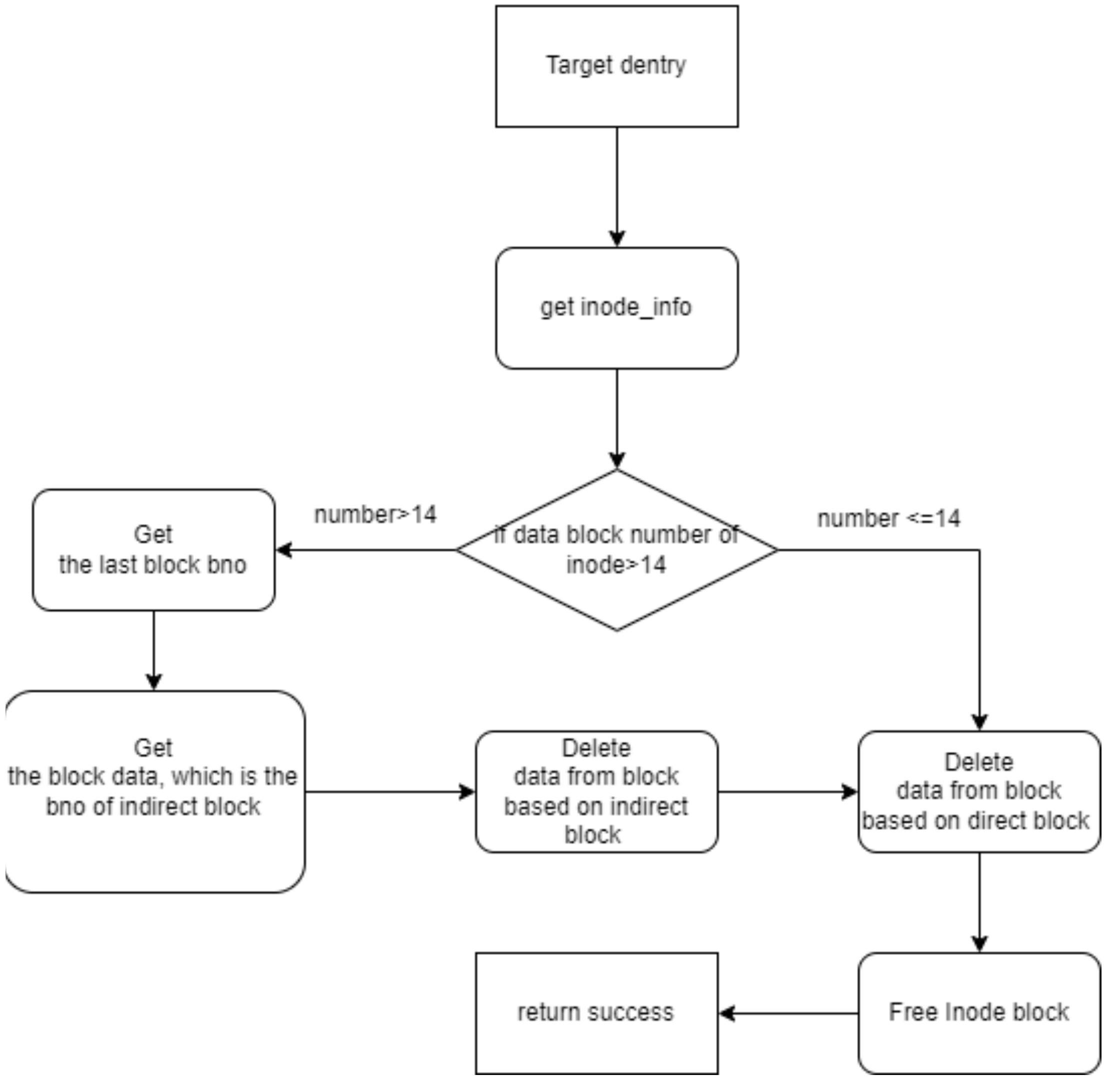

Target dentry
get inode_info
if data block number of inode>14
number>14
number <=14
Get the last block bno
Get the block data, which is the bno of indirect block
Delete data from block based on indirect block
Delete data from block based on direct block
Free Inode block
return success

Figure 7: steps to delete a file with indirect pointer.

## 5.4 Creation and access of indirect blocks

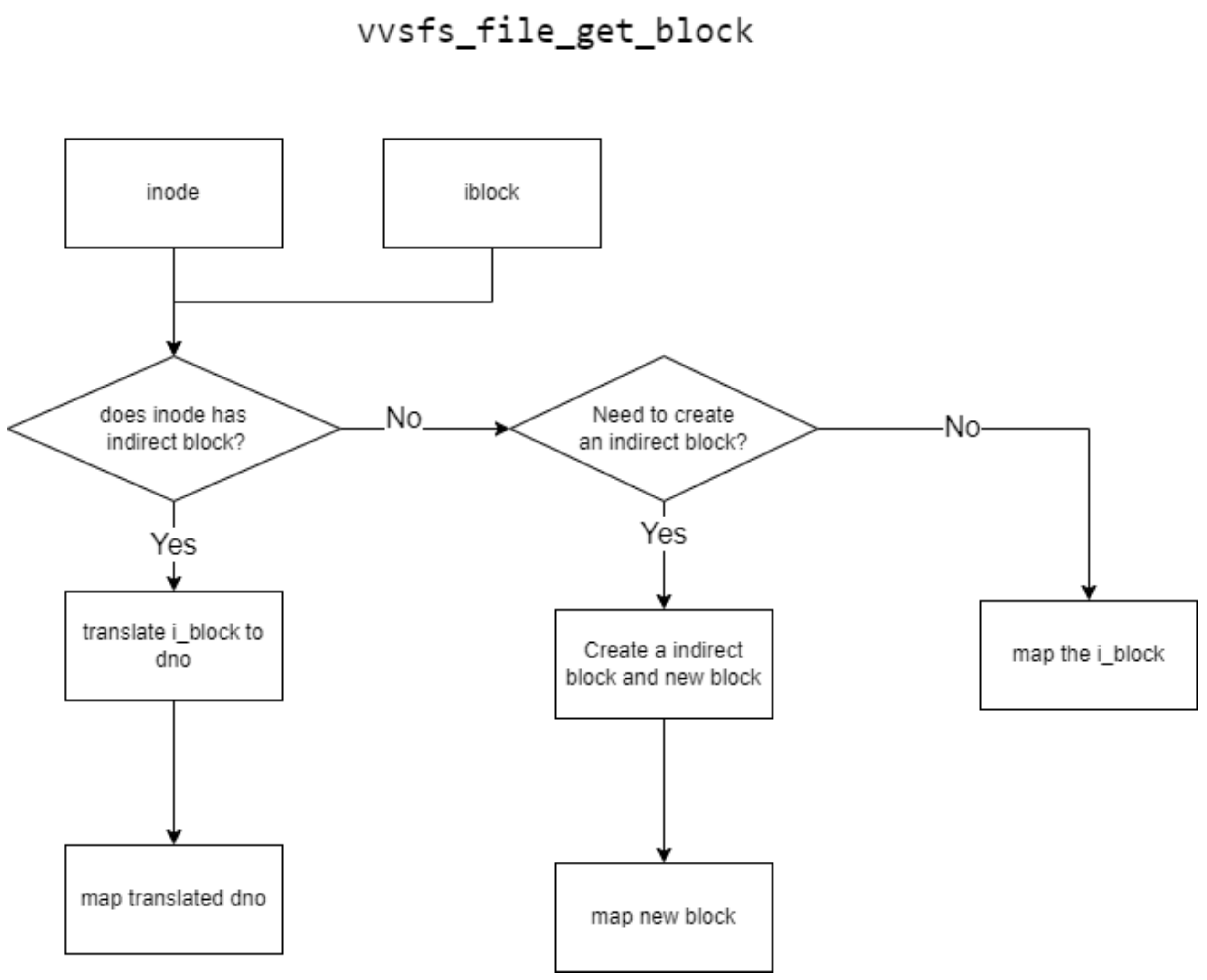


Figure 8: flowchart vvsfs_file_get_block

First, our implementation uses generic file operation in the struct file_operations. Those operations are only depending on address space operations. We notice that the address space operations use vvsfs_file_get_block function to determine the buffer that the input data needed to be written into.

Thus, we modify the implementation of vvsfs_file_get_block so that it is aware of indirect blocks.

As shown in Figure 8, in this modified version function, when we are requested to map a block that might be indicated block. We first check whether the ‘indirect block’ is created. If the indirect block is created, we reserved the data block and put the block number into the right place in the indicted block.

Otherwise, we will reserve a data block, as the indirect block. And reserve another data block, as the actual block for storing content.

# 6 Reflection

## 6.1 Reflect on Challenges and Solutions

During this process, the biggest difficulty we encountered was the lack of documentation in the program. We always stared at the code, trying to guess how to use functions or structures. Occasionally, you will find some scattered documents, but these documents are not enough to form a large framework of the entire file system.

1. Carefully checked the documents in the homework and laboratory, and finally found three links with relatively complete documents, which contain the source code for the use of VFS and the implementation of similar file management systems. This discovery has given us a lot of help.

2. Search the Internet for related functions and usage of structures. Unfortunately, due to the lack of popularity of relevant information on the Internet, we were not able to find enough useful relevant technical documentation. The Internet resource that has contributed the most to us is this article on how to switch kernel versions, which has been of great help to our group of students who have different kernel versions.

## 6.2 Reflect on Testing

The main way to test and debug our problem is to use “printk” to observe the behaviour of the problem during running. We have also done manual testing, that is, create, delete, and copy files in the file system and use “ls”, “df”, and “stat” to observe the changes. For example, if all files are deleted, does “ls” print nothing and the block used returned by “df” decrease back to 2 (block used by the root folder). We also write some automatic scripts to make the compile process faster and more convenient.

# 7 Acknowledgement

Scripts:

We employ the pre-commit hooks configuration file[2] to enforce code quality checks, and we utilize Git tools to manage our code.

[2] [2]“pre-commit,” pre-commit.com. https://pre-commit.com/

*We declare that everything we have submitted in this assignment is entirely our own work, with the following exceptions:*

- *list the sources of code you used, articles, blogs, books etc, if any*
- *discussions with others related to this assignment, if any*