

Project 3: Design and Implementation of File System

UNikeEN

May 14, 2023

Contents

1	Step 1: Design a Basic Disk-storage System	3
1.1	Implementation	3
1.1.1	Simulated Disk Structure	3
	Struct Definition	3
	Initialize Disk Struction	3
	Destructor	4
1.1.2	Mapping	4
1.1.3	Command Modularity	5
	Struct Definition	5
	Classification and Execution	5
1.1.4	Command Implementation	6
	Read Data (R-command)	6
	Write Data (W-Command)	7
	Information Request and Exit (I and E-Command)	7
1.2	Running Result	7
2	Step 2: Design a Basic File System	9
2.1	Design	9
2.1.1	Architecture Overview	9
2.1.2	Code Structure	10
2.1.3	Data Structure	11
	Superblock	11
	I-node and Dir Item	12
2.2	Implementation	13
2.2.1	Simulated Disk	13
2.2.2	Superblock	14
	Structure Definition	14

Bitmap, Allocation and Recycling	14
2.2.3 Directory-type I-node	15
Structure Definition	15
Add Subitem	16
Delete Subitem	17
List Subitems	18
2.2.4 File-type I-node	18
Read File Content	18
Write File Content	18
Append, Delete and Insert Content	20
2.2.5 Frontend Shell	21
Command Modularity	21
2.3 Running Result	22
3 Step 3: Work Together	23
3.1 Implementation	23
3.1.1 Socket Programming	23
Connect and Listen	23
Online Disk Operation	25
Response Display	26
Key Points in Data Transmission	26
3.1.2 Persistence and Retrieval of File System	27
Magic Number	27
Adaptation of Disk Size	27
3.1.3 Log Detailization	28
3.2 Running Result	28
4 Further Development and Prospect	29
4.1 Multi-Client Support	29
4.2 Multi-User Support and Authority Management	30
4.3 Directory Cache and Write-back Strategy	30
5 Conclusion	31
References	31

1 Step 1: Design a Basic Disk-storage System

1.1 Implementation

1.1.1 Simulated Disk Structure

Struct Definition Firstly, I use a `struct` called 'SimDisk' to define the various information of the disk. Here is the code:

```
typedef struct{
    char *filename;
    int fd;
    int num_cylinders;
    int sectors_per_cylinder;
    int track_to_track_delay;
    char *diskfile;
    FILE *log_fp;
} SimDisk;
```

The following are explanations for each member variable:

- `filename`: The name of the real disk file that stores data.
- `fd`: The file descriptor of the above file.
- `num_cylinders`, `sectors_per_cylinder`: From these two integers, the storage capacity of the disk can be obtained. Assuming a sector size of 256 bytes. The storage capacity in total is $\text{num_cylinders} \times \text{sectors_per_cylinder} \times 256$ bytes.
- `track_to_track_delay`: Simulate the delay time of disk head movement between tracks.
- `diskfile`: An array that stores disk data in memory.
- `log_fp`: Pointer to the file stream, used to open and write log files.

Initialize Disk Struction Based on the guidance of slides, I have implemented the following function to initialize the disk structure. Open the sim-disk file and the log file, use the system call `lseek()` to stretch the file size to the size of the simulated disk, and try to Write something at the end of the file to ensure the file actually have the new size.

If each system call cannot return the correct value, the program will print a warning. Here is the code:

```
void Init(SimDisk *disk, int num_cylinders, int sectors_per_cylinder, int
    track_to_track_delay, const char *filename){
    disk->num_cylinders = num_cylinders;
```

```
disk->sectors_per_cylinder = sectors_per_cylinder;
disk->track_to_track_delay = track_to_track_delay;
disk->filename = strdup(filename);

disk->fd = open(filename, O_RDWR | O_CREAT, 0666);
if (disk->fd < 0) ...//print error in calling open()

// Calculate the disk storage size
long filesize = num_cylinders * sectors_per_cylinder * SECTOR_SIZE;

int result = lseek(disk->fd, filesize - 1, SEEK_SET);
if (result != 1) ...//print error in calling lseek() to 'stretch' the file

result = write(disk->fd, "", 1);
if (result != 1) ...//print error in writing last byte of the file
...
disk->log_fp = fopen("disk.log", "w");
}
```

Destructor Similar to the class in C++, when designing the constructor (as above), the destructor should also be designed to close each file descriptor and free space for dynamic allocation. When the program ends, this function will be called.

1.1.2 Mapping

Based on the guidance of slides, I use the `mmap()` system call to create a mapping between a memory space and a file. Replacing I/O reads and writes with memory reads and writes can achieve high performance.

Here is the code of creating mapping in `Init()`:

```
// Map the disk storage file to memory
disk->diskfile = mmap(NULL, filesize, PROT_READ | PROT_WRITE, MAP_SHARED,
    disk->fd, 0);
if (disk->diskfile == MAP_FAILED)
{
    close(disk->fd);
    printf("Error: Could not map file.\n");
    exit(-1);
}
```

Here is the code of removing mapping in `Destructor()`:

```
// Unmap the disk storage file from memory
if (munmap(disk->diskfile, disk->num_cylinders * disk->sectors_per_cylinder *
    SECTOR_SIZE) == -1)
{
    perror("Error unmapping disk storage file");
}
```

1.1.3 Command Modularity

Struct Definition In this step, the most intuitive way to distinguish commands is to **switch** the first char of the input string. But inspired by the ideas of base classes and inheritance in C++, I treated commands as modules, simulated base class with **struct**, and defined the structures that command modules need to follow. The following is the code for the its definition:

```
typedef struct Command {
    bool (*judge_trigger)(const char* cmd_head);
    int (*handle)(SimDisk *disk, int cylinder, int sector, char* data);
} Command;
```

The member variable is two function pointers, defined as follows:

- **judge_trigger**: This function returns a `bool` variable to identify the command.
- **handle**: This function contains the execution process of the command, and the integer returned represents the status code passed back to the main program loop.

Classification and Execution I used an array to store the "command module" structure representing different instructions. In this step, for each command inputted, the program first uses `sscanf()` for parsing, and then loops through the '`judge_trigger()`' function of each 'command module' in the array for format matching. Once matched, the '`handle()`' function of the 'command module' is executed to exit the traversal. If the array is not successfully matched after traversing, it indicates that the command format is incorrect.

Here is the code:

```
bool I_judgeTrigger(const char* cmd_head){ return cmd_head[0]=='I'; }
bool R_judgeTrigger(const char* cmd_head){ return cmd_head[0]=='R'; }
bool W_judgeTrigger(const char* cmd_head){ return cmd_head[0]=='W'; }
bool E_judgeTrigger(const char* cmd_head){ return cmd_head[0]=='E'; }

Command commands[] = {
```

```

    {I_judgeTrigger, I_handle},
    {R_judgeTrigger, R_handle},
    {W_judgeTrigger, W_handle},
    {E_judgeTrigger, E_handle},
};

const int num_commands = sizeof(commands) / sizeof(Command);

//In main()
while (1){
    fgets(raw_command, sizeof(raw_command), stdin);

    bool handled = false;
    for (int i = 0; i < num_commands; i++) {
        cylinder = -1; sector = -1;
        sscanf(raw_command, "%s %d %d %[^\\n]", cmdtype, &cylinder, &sector, data);
        if (commands[i].judge_trigger(cmdtype)) {
            ret = commands[i].handle(&disk, cylinder, sector, data);
            handled = true;
            break;
        }
    }
    if (!handled) {
        printf("Undefined command!\\n");
    }
    if (ret == -1) break; //E command means exit
}

```

Modularization has little impact on performance, but it is easy to maintain, add or delete. For more complex command formats and more commands in subsequent steps, it is important to modularize commands. **This step is only to verify their realizability.**

1.1.4 Command Implementation

The following will introduce the response implementation of each instruction in the protocol, namely the 'handle()' function of each 'command module'.

Read Data (R-command) First, check the legitimacy of the address, then output the stored characters one by one from the corresponding address until the end of the data or the end of the sector is reached.

Here is the code:

```
int R_handle(SimDisk *disk, int cylinder, int sector, char* data){
    ...//warning the invalid address
    char *sector_data = disk->diskfile + (cylinder * disk->sectors_per_cylinder +
        sector) * SECTOR_SIZE;
    fprintf(disk->log_fp, "Yes ");
    int i = 0;
    while (sector_data[i]){
        fprintf(disk->log_fp, "%c", sector_data[i]);
        i++;
        if (i == SECTOR_SIZE) break;
    }
    fprintf(disk->log_fp, "\n");
}
```

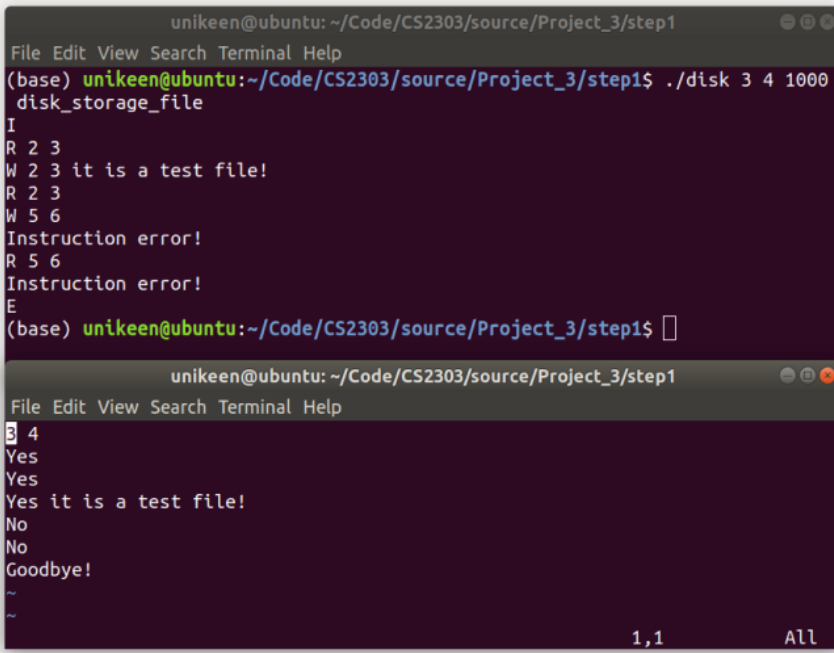
Write Data (W-Command) First, check the legitimacy of the address, and then use the system call `memcpy` to copy the input data to the corresponding location in memory with the size of the sector.

```
int W_handle(SimDisk *disk, int cylinder, int sector, char* data){
    ...//warning the invalid address
    char *sector_data = disk->diskfile + (cylinder * disk->sectors_per_cylinder +
        sector) * SECTOR_SIZE;
    memcpy(sector_data, data, SECTOR_SIZE);
    fprintf(disk->log_fp, "Yes\n");
    return 0;
}
```

Information Request and Exit (I and E-Command) The implementation of these two commands is relatively simple, where the exit command's '`handle()`' will return a status code of **-1** to prompt the main function to exit.

1.2 Running Result

Here are some screenshots of the runtime, please refer to the video for a detailed demonstration.



```
unikeen@ubuntu: ~/Code/CS2303/source/Project_3/step1
File Edit View Search Terminal Help
(base) unikeen@ubuntu:~/Code/CS2303/source/Project_3/step1$ ./disk 3 4 1000
disk_storage_file
I
R 2 3
W 2 3 it is a test file!
R 2 3
W 5 6
Instruction error!
R 5 6
Instruction error!
E
(base) unikeen@ubuntu:~/Code/CS2303/source/Project_3/step1$

unikeen@ubuntu: ~/Code/CS2303/source/Project_3/step1
File Edit View Search Terminal Help
4
Yes
Yes
Yes it is a test file!
No
No
Goodbye!
~
1,1 All
```

Fig. 1. Screenshot of Step 1

2 Step 2: Design a Basic File System

2.1 Design

2.1.1 Architecture Overview

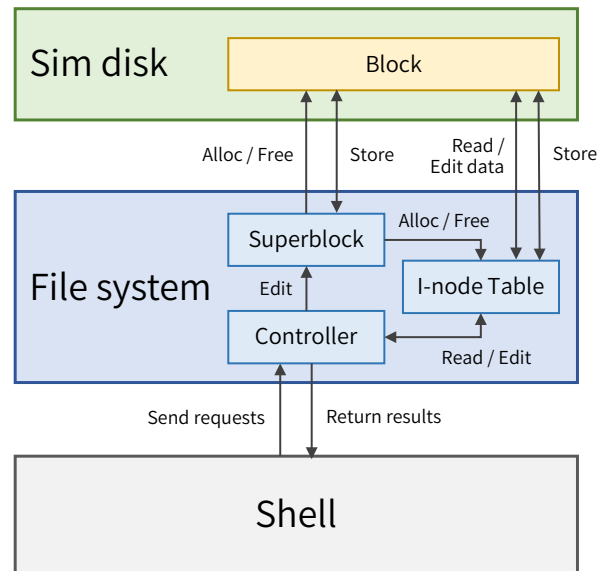


Fig. 2. Architecture Overview

As shown in Fig. 2, the file system consists of three parts: a user oriented **shell**, an **i-node based core** part, and a **simulated disk**.

- **Shell:** Similar to the shell implemented in project 1, the shell in this experiment works in a loop, receiving user input, recognizing and processing commands.
- **I-node Based Core:** The core of the file system draws inspiration from Linux's EX2 file system[1], using the ideas of **i-node** and **superblock**. Each file and directory is associated with a unique i-node, containing metadata about the file or directory; Superblock stores the overall information of the file system and manages the allocation or recycling of i-nodes.
- **Simulated Disk:** Similar to the disk system in step 1, in this step, we also established an array to simulate the disk to store data. All data is read or stored in blocks, and the superblock and i-nodes are also aligned and stored in specific numbered blocks, while the specific data of the file content is stored in other blocks with uncertain numbers and dynamically allocated by the superblock. This implementation also provides convenience for the next step 3.

2.1.2 Code Structure

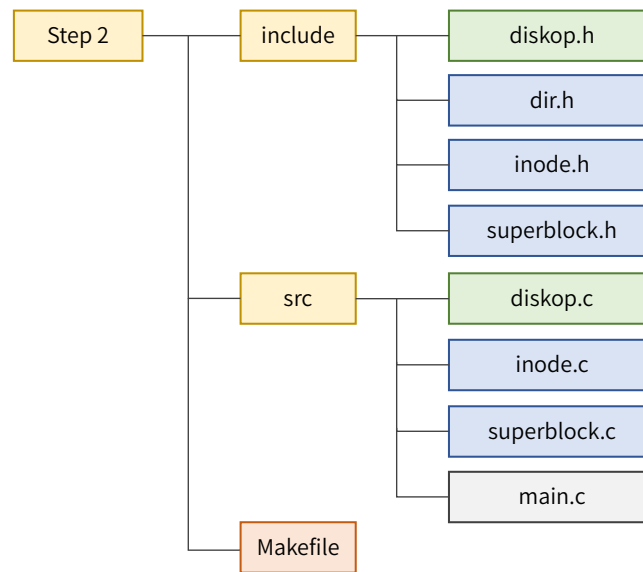


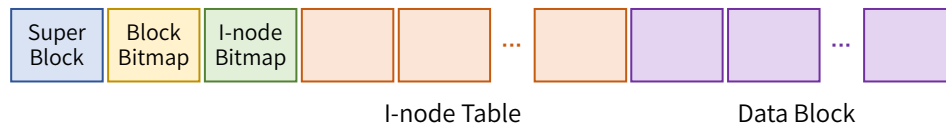
Fig. 3. Code Structure

This step involves a significant amount of code engineering, so I have established the code structure shown in Fig. 3 using the concept of modular programming. The content of each file is as follows:

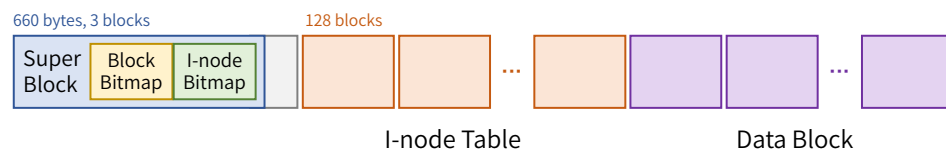
- **diskop.h** and **diskop.c**: Defined an array `disk` for storing all data (`disk_File`), providing operation functions for reading and writing by block.
- **inode.h** and **inode.c**: Defined the core data structure `inode`, which provides basic functions such as allocating and returning idle i-nodes, as well as reading and editing i-nodes; It also provides functions for searching by name, adding and deleting subitems for directory type i-nodes, as well as functions for reading and writing the contents of file type i-nodes in blocks.
- **dir.h**: Defined a data structure called `dir_item`, including the names and i-node numbers of subdirectories and files. Through it, a tree directory structure has been implemented.
- **superblock.h** and **superblock.c**: The data structure of the superblock has been defined to store basic information about the file system. Recorded the current idle i-node and block numbers, providing functions for allocating and returning idle blocks.
- **main.c**: An implementation of a self-made shell that stores and displays the current path of the file system, and modularizes commands according to step 1. Receive user input, recognize commands, and call the functions in the above library to process them, then display the results.
- **Makefile**: Use `gcc` and `C11` standard to link, compile, and build the entire program, with the target executable file named `fs`.

2.1.3 Data Structure

As mentioned above, all the information of the file system is stored in blocks within a virtual disk array. The following are the respective implementations in the guide slides and my actual implementation.



(a) From Guide Slides



(b) My Actual Implementation

Fig. 4. Block Array

Superblock The superblock contains essential information about the file system, such as the total and available number of i-nodes and blocks, the i-node number of the root directory, and two bitmap arrays used to store the availability of blocks and i-nodes. Each integer in the bitmap is converted to a 32-bit binary representation, corresponding to the block and i-node numbers, where 0 represents free and 1 represents occupied. When creating or deleting files and directories, the allocation or deallocation of i-nodes is required. Similarly, for writing and reading data, the allocation or deallocation of blocks is necessary. These operations are managed by synchronously updating the two bitmaps in the superblock.

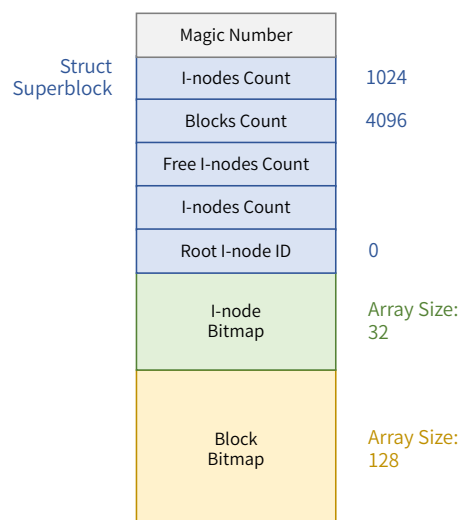


Fig. 5. The struct of Superblock

Shown in Fig. 5, during instantiation, the aforementioned quantities and bitmap size are fixed, representing a fixed size for the sim disk. Therefore, if this code is used without modification for step 3, there will be a lower and an upper limit on the supported virtual disk.

The actual superblock may also include a magic number to identify the file system, but it is not implemented in this experiment as it is not relevant.

I-node and Dir Item Each i-node uniquely corresponds to a directory or file and stores information such as the type (0 for file, 1 for directory), timestamps, link count and etc. Its size is 32 bytes. When assuming a block size of 256 bytes, each disk block in the i-node table can accommodate 8 i-nodes. The size of each block in Fig. 6 represents the size of the data type (e.g., the minimum mode and link count are represented by `uint8_t`).

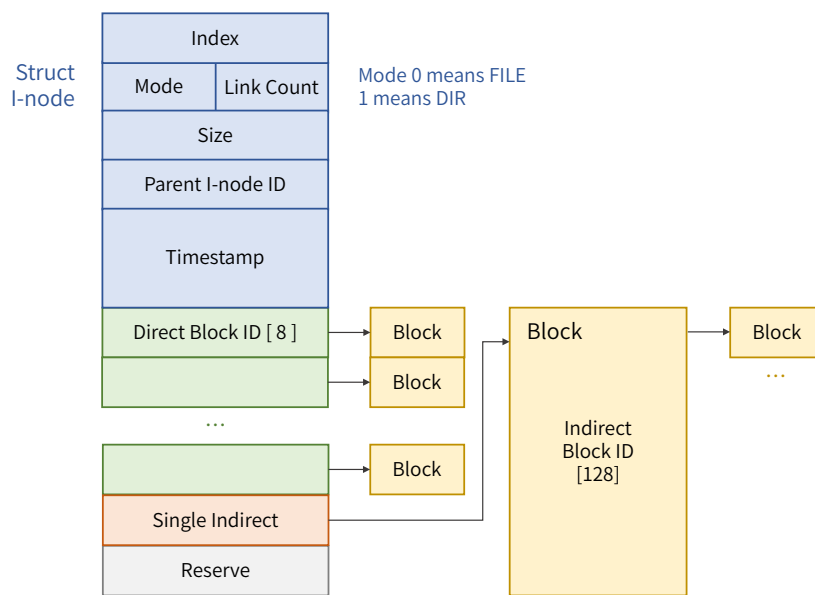


Fig. 6. The struct of I-node

The i-node implementation I have developed includes 8 direct links, which store the corresponding block numbers of the data. Additionally, there is one indirect link that points to a block containing the numbers of 128 more data blocks. So that **file-type** i-node can store a maximum of 136 blocks of data, When data needs to be read or written, we can retrieve the data from the corresponding data block based on its assigned block number.

On the other hand, a **directory-type** i-node only uses the direct links section, with each corresponding block capable of storing 8 directory entries (stored by structure `dir_item` in Fig. 7. This structure contains the name, valid flag, and inode number of a file or directory. The file system can open the corresponding inode by traversing the names and types of valid entries when reading, writing files, or changing paths. As a result, first-level subfiles or subdirectories under the same directory cannot have the same name. The file system checks for this when creating new files or directories.). Therefore, a directory can have a maximum of 64 directory entries, including up to

64 first-level subdirectories or files.

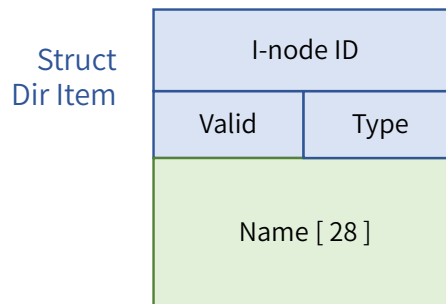


Fig. 7. The struct of Dir Item

Additionally, directory-type i-nodes can use “parent” field that stores the inode number of the parent directory. This allows for upward navigation when changing paths. The parent of the root directory is predetermined with a value of 65535.

Furthermore, 2 bytes are reserved for position, which can be used to store information such as the next available block or for second-level indirect linking.

2.2 Implementation

2.2.1 Simulated Disk

In this experiment, this is the lowest-level data structure, stored as an array. The definition code for this structure is as follows:

```
char disk_file[MAX_BLOCK_COUNT][BLOCK_SIZE];
```

The read and write operations for blocks are relatively straightforward and can be accomplished using the `memcpy()` function. Moreover, The implemented write function supports writing to consecutive multiple blocks. Here are the codes:

```
int write_block(int block_no, char *buf, int block_num){
    ...//Check the legality of the block number
    for (int i = 0; i < block_num; i++)
        memcpy(disk_file[block_no + i], buf + BLOCK_SIZE * i, BLOCK_SIZE);
    return 0;
}

int read_block(int block_no, char *buf){
    ...//Check the legality of the block number
    memcpy(buf, disk_file[block_no], BLOCK_SIZE);
    return 0;
}
```

2.2.2 Superblock

Structure Definition According to Section 2.1.3 and Fig. 5, the definition code is as follows:

```
typedef struct super_block{
    uint32_t s_inodes_count;
    uint32_t s_blocks_count;
    uint32_t s_free_inodes_count;
    uint32_t s_free_blocks_count;
    uint32_t s_root;
    uint32_t block_map[MAX_BLOCK_MAP];
    uint32_t inode_map[MAX_INODE_MAP];
} super_block; //660 bytes, 3 blocks
```

The total size of the superblock is 660 bytes, assuming a block size of 256 bytes, which requires 3 blocks when rounded up. As depicted in Fig. 4, we define that the superblock is stored in blocks numbered 0, 1, and 2. The following code is used to synchronously write the modifications of superblock to the simulated disk:

```
char buf[3*BLOCK_SIZE]; memset(buf, 0, sizeof(buf));
memcpy(buf, &spb, sizeof(spb));
if(write_block(0, buf, 3) < 0) return -1;
```

Bitmap, Allocation and Recycling According to Section 2.1.3, In the bitmap array, each 32-bit binary number represents the availability status of 32 blocks or inodes.

For example, the 32 numbers in `block_bitmap[3]` represent blocks 128-159. Initially, the superblock and inode table occupy 130 blocks, which are assigned a value of `0xe0000000`, corresponding to `1110000000..00` in binary, indicating that blocks 128-130 are already occupied.

```
for(int i = 0; i <= 3; i++)
    spb.block_map[i] = ~0; //Bits 1-131 occupied, block_map Write 1
spb.block_map[4] = (0xe0000000);
```

The superblock manages allocation and deallocation through the bitmap. Taking block allocation as an example, during traversal, each bit is read using a right-shift operation. The availability is determined by performing a bitwise AND operation, and the selected block is modified to 1 (indicating occupied) using a bitwise OR operation. Here is the code:

```
for (int i = 4; i < MAX_BLOCK_MAP; i++) {
    uint32_t block = spb.block_map[i];
    for (int j = 0; j < 32; j++){
        if ((block >> (31-j)) & 1) continue; //block is not empty
```

```

        else {
            spb.s_free_blocks_count--;
            spb.block_map[i] |= 1 << (31-j); //change bitmap to 1
            ...
            return i*32+j;
        }
    }
}

```

The deallocation process is relatively simple and does not require a loop traversal. Similarly, the specified bit is read using a right-shift operation, and the selected block is modified to 0 (indicating free) using a bitwise XOR operation. Here is the code:

```

int i = index / 32; int j = index % 32;
if (((spb.block_map[i] >> (31 - j)) & 1) == 0) return 1; //block is already idle
else{
    spb.block_map[i] ^= 1 << (31-j); //change bitmap to 0
    spb.s_free_blocks_count++;
    ...
}

```

The allocation and recycling process of inodes follows a similar approach, with the only difference being the size of the bitmap.

2.2.3 Directory-type I-node

Structure Definition According to Section 2.1.3 and Fig. 6 and Fig. 7, the definition code of inode and dir_item is as follows:

```

typedef struct inode {
    uint8_t i_mode;           // 0 represents file, 1 represents directory
    uint8_t i_link_count;     // Number of existing links
    uint16_t i_size;          // Records file size
    uint32_t i_timestamp;     // Records file modification time
    uint16_t i_parent;        // Parent inode number (65535 represents none)
    uint16_t i_direct[8];     // Direct blocks (record block numbers, block
                             // numbers use uint16_t uniformly)
    uint16_t i_single_indirect; // Single indirect block
    uint16_t reserve;
    uint16_t i_index;
} inode; // 32 bytes
inode inode_table[MAX_INODE_COUNT];

```

```
typedef struct dir_item {
    uint16_t inode_id;      // Inode corresponding to the file/directory
                           // represented by the current directory entry
    uint8_t valid;          // Whether the current directory entry is valid
    uint8_t type;           // Type of the current directory entry
                           // (file/directory)
    char name[MAX_NAME_SIZE]; // File name/directory name of the file/directory
                           // represented by the directory entry
} dir_item; // 32 bytes
```

Each inode occupies 32 bytes, and an inode table with 128 blocks can store a maximum of 1024 inodes.

Based on Section 2.1.3 and Fig. 6 and 7, the `i_mode` field of an inode is set to 1, indicating that it is a directory. Each direct link of the inode points to a block that contains 8 `dir_item` structures.

Add Subitem The process of adding a subdirectory or subfile is the same. Firstly, the existing direct link blocks are checked to see if there is any available space (corresponding to `dir_item` with `valid = 0`). If there is no available space and there are still direct links without allocated blocks (indicated by `inode.i_indirect = 0`), a new block is allocated. If both conditions are not met, it indicates that the limit has been reached (the number of first-level subdirectories and subfiles is less than or equal to 64). The following is the core code snippet:

```
char buf[BLOCK_SIZE];
dir_item dir_items[8];
for (int i = 0; i < 8; i++) { // Traverse 8 direct blocks
    if (dir_node->i_direct[i] == 0) continue;
    ...
    memcpy(&dir_items, buf, BLOCK_SIZE);
    for (int j = 0; j < 8; j++) { // Traverse 8 dir_items in the block and find
        an empty slot
        if (dir_items[j].valid == 0) {
            int new_inode_id = alloc_inode(); // Allocate a new inode
            if (new_inode_id >= 0) {
                init_inode(&inode_table[new_inode_id], new_inode_id, type, 0, 0,
                    dir_node->i_index); // Initialize the new inode
                ... // Modify dir_item and current directory inode information,
                    write back to disk
            }
            ...
        }
    }
    if (dir_node->i_link_count < 8) { // If the above process has no result but link
```



```

    count is less than 8, allocate a new block
    int k;
    for (k = 0; k < 8; k++) {
        if (dir_node->i_direct[k] == 0) break;
    }
    int new_block_id = alloc_block();
    if (new_block_id >= 0) {
        int new_inode_id = alloc_inode(); // Allocate a new inode
        if (new_inode_id >= 0) {
            dir_node->i_direct[k] = new_block_id; // Allocate a new block
            dir_node->i_link_count += 1;
            for (int i = 0; i < 8; i++) {
                dir_items[i].valid = 0;
            }
            init_inode(&inode_table[new_inode_id], new_inode_id, type, 0, 0,
                dir_node->i_index); // Create a new inode for the new direct block
            ... // Modify dir_item and current directory inode information, write
                back to disk
        }
    }
    ...

```

Please note that we specify that file and directory names should be less than 28 bytes and can only contain alphanumeric characters, underscores, and periods. Within the same directory, files or subdirectories cannot have the same name, although a file and a directory can have the same name. The former is checked by a frontend function, while the latter is handled by another function, `search_in_dir_inode()`, called from the frontend, which searches for and returns the inode number based on the name in `inode.h`.

Delete Subitem The principles of deleting files and folders are nearly identical. The process involves traversing the `dir_item` structures of the current directory's inode based on the name and searching for the target file or folder. For files, simply deallocating their respective inode is sufficient.

However, for folders, similar to Linux commands, it is necessary to check if subfolders are empty (i.e., all `dir_item.valid = 0`). If the folder is not empty, a prompt is displayed, and the operation is terminated. Recursive or forced deletion functionalities are not provided. The following is the core code snippet:

```

char buf[BLOCK_SIZE];
dir_item dir_items[8];
for (int i = 0; i < 8; i++) {
    if (dir_node->i_direct[i] == 0) continue;

```

```

if (read_block(dir_node->i_direct[i], buf) < 0) continue;
memcpy(&dir_items, buf, BLOCK_SIZE);
for (int j = 0; j < 8; j++) {
    if (dir_items[j].valid == 0) continue;
    if (strcmp(dir_items[j].name, name) == 0 && dir_items[j].type == 1) {
        // Traverse each item, check the name and corresponding inode's i_mode.
        // If the name matches and it is a folder/file:
        ...// If it is a folder, check if it is empty
        free_inode(&inode_table[dir_items[j].inode_id]); // Return the inode
        ...// Modify dir_item and current directory inode information, write
            back to disk
    }
}
...
}

```

List Subitems Corresponding to the `ls` command in the shell, its principle involves traversing the valid `dir_item` structures of the current directory's inode. It retrieves the names of subfiles and subdirectories, applies the `qsort()` function from the `stdlib.h` library to sort them in **ascending lexicographical order**, and then outputs the results to the shell (with different colors to distinguish files and directories) and a log file (differentiated by the use of '&' as required).

2.2.4 File-type I-node

Based on Section 2.1.3 and Fig. 6, the `i_mode` field of an inode is set to 0, indicating that it is a file. Direct links can store 8 block numbers. Each file inode can occupy a total of $8 + 1 + 128 = 137$ blocks, where 136 blocks correspond to the file content and 1 block is pointed to by the indirect link, storing 128 block numbers.

Read File Content Reading file content is relatively simple. By traversing the data blocks stored by the file's inode, the data can be concatenated. There are some challenges that may arise during this process. When multiple blocks are involved, the block data may not include the string termination character. Therefore, using `memcpy()` is preferable over `strcpy()` for concatenation. Additionally, it is essential to clear the variables (using `memset()`) used for passing data each time before their use.

Write File Content The term "write" here refers to overwrite writing. The simplest approach is to recycle all the data blocks of the file and then allocate new data blocks based on the length of the written content. However, this method incurs significant unnecessary performance overhead

when dealing with large data. To address this, I have adopted a different approach where I retain the existing blocks and, based on the required number of blocks for the written content, deallocate excess blocks or allocate new blocks as needed.

Additionally, before allocating new blocks, the information in the superbloc (spb) regarding the number of available blocks can be compared. If there are insufficient blocks, the process can be terminated early to avoid issues with deallocating already allocated blocks if the allocation process is interrupted midway. Here are the core code snippet:

```
int write_file_inode(inode* file_node, char* src){
    if (file_node->i_mode) return -1; // Not a file, exit
    uint16_t indirect_blocks[128];
    int need_link_count = (strlen(src) + BLOCK_SIZE - 1) / BLOCK_SIZE;
    int ori_link_count = file_node->i_link_count; // Calculate the new and
        original required block counts
    ...
    if (need_link_count <= ori_link_count){ // Sufficient existing blocks
        // Write content data
        for(int i = 0; i < (need_link_count>=8 ? 8:need_link_count); i++){
            memcpy(buf, tmp + i*BLOCK_SIZE, BLOCK_SIZE);
            write_block(file_node->i_direct[i], buf, 1);
        }
        if (read_block(file_node->i_single_indirect, buf) < 0) return -1;
        memcpy(&indirect_blocks, buf, BLOCK_SIZE);
        for(int i = 0; i < (need_link_count - 8); i++){
            memcpy(buf, tmp + (i+8)*BLOCK_SIZE, BLOCK_SIZE);
            write_block(indirect_blocks[i], buf, 1);
        }
        // Return excess blocks
        for(int i = need_link_count; i < (ori_link_count>=8 ? 8:ori_link_count);
            i++){
            free_block(file_node->i_direct[i]); // Excess blocks linked directly
            file_node->i_direct[i] = 0;
        }
        if (ori_link_count > 8 && file_node->i_single_indirect!=0){ // Excess
            blocks linked indirectly
            for(int i = (need_link_count<8?0:need_link_count-8); i <
                ori_link_count - 8; i++){
                free_block(indirect_blocks[i]);
                indirect_blocks[i] = 0;
            }
        }
    }
}
```

```

        } ...// Empty or modify and write back indirect links
    ...
    else{// Allocate new nodes for insufficient blocks
        // Compare the required new blocks with spb, if insufficient, exit
        if (need_link_count - ori_link_count > spb.s_free_blocks_count) return -1;
        for(int i=ori_link_count; i<(need_link_count > 8 ? 8 : need_link_count);
            i++){ // Allocate new blocks for direct links if available
            uint16_t new_block_id = alloc_block();
            if (new_block_id < 0) return -1;
            file_node->i_direct[i] = new_block_id;
        }
        if (need_link_count > 8) ...// Allocate indirect link blocks (if
            necessary) and allocate new blocks within indirect links
        ...// Write content data
    } ...// Update inode information and write back to block
    return 0;
}

```

Append, Delete and Insert Content Operations such as appending, deleting, or inserting characters within the content do not require the implementation of new functions. Instead, these operations can be achieved by combining the aforementioned read and write functions in the frontend (shell) of the system.

For example, the core code for the insertion operation in `main.c` is as follows, Error capture has been omitted:

```

read_file_inode(&inode_table[ret_id], buf); // Read existing data
...
memmove(buf + pos + len, buf + pos, file_size - pos); // Move data backward
memcpy(buf + pos, src, len); // Insert data
buf[new_size] = '\0';
write_file_inode(&inode_table[ret_id], buf); // Write data

```

Please note that in the implementation of step 2, we assumed that reformatting is required each time the system is started. Therefore, for changes made to inodetable, superbblock, and other components, we only implemented the synchronous write-back operation to the simulated disk when the changes occur, **but we did not implement the read operation from the simulated disk.** By adding the read operation (**added in step 3, discussed in Section 3.1.2**), we can achieve the functionality of loading an existing file system during startup.

2.2.5 Frontend Shell

Command Modularity Similar to what was described in Section 1.1.3, a structure is used to simulate a base class, with standardized definitions of command structures that include trigger words and function pointers for execution. When receiving user input, the command list is traversed, and if a trigger word matches, the corresponding execution function is executed.

```
typedef struct Command {
    char cmd_head[6];           // Command name, used for matching the
                                // beginning of each input line
    int (*handle)(char* params); // Function to handle the command
} Command;

Command commands_list[] = {
    {"f", format_handle},
    {"mk", mk_handle},
    {"mkdir", mkdir_handle},
    ...
    {"w", w_handle},
    {"i", i_handle},
    {"d", d_handle},
    {"gb", debug_getb_handle}, // DEBUG
    {"gi", debug_geti_handle}, // DEBUG
    {"e", exit_handle},
};

const int num_commands = sizeof(commands_list) / sizeof(Command);
... // Iterate through and match the command name (trigger word) during execution
for (int i = 0; i < num_commands; i++) {
    sscanf(raw_command, "%s %[^\\n]*c", cmd_head, params);
    if (strcmp(cmd_head, commands_list[i].cmd_head) == 0) {
        ret = commands_list[i].handle(params);
        handled = true;
        break; // Exit the loop if the command is successfully executed
    }
}
```

Regarding each command, existing functions from other libraries can be called to perform the operations. For example, the “f” command used for formatting the file system may not have a dedicated “format” function, but it can be implemented by invoking functions such as `init_spb()`, `init_inode()` and `init_root_inode()`. For operations like inserting or deleting characters in the

content, as described in Section 2.2.4, functions like `read_file_inode()`, `write_file_inode()` and standard functions like `memcpy()` and `memmove()` can be utilized.

Importantly, as for the “cd” command, in addition to traversing to find subdirectories and change the current dir-inode, the frontend shell also maintains a string that stores and displays the absolute address of the current directory within the file system.

The details of other commands are omitted here.

2.3 Running Result

Here are some screenshots of the runtime, please refer to the video for a detailed demonstration.

```

unikeen@ubuntu: ~/Code/CS2303/source/Project_3/step2
File Edit View Search Terminal Help
(base) unikeen@ubuntu:~/Code/CS2303/source/Project_3/step2$ ./fs
UnikeEN-fs: >> f
UnikeEN-fs:/ >> mkdir src
UnikeEN-fs:/ >> mkdir lib
UnikeEN-fs:/ >> mk #tmp
Illegal name! (Only accept numbers ,letters ,_ and . )
UnikeEN-fs:/ >> mk tmp
UnikeEN-fs:/ >> ls
tmp lib src
UnikeEN-fs:/ >> cd ./src
UnikeEN-fs:/src >> mk a.cpp
UnikeEN-fs:/src >> w a.cpp 8 #include<iostream>
Warning: The string length parameter does not match the actual length!
UnikeEN-fs:/src >> cat a.cpp
#include<iostream>
UnikeEN-fs:/src >> d a.cpp 3 5
UnikeEN-fs:/src >> cat a.cpp
#include<iostream>
UnikeEN-fs:/src >> cd ..
UnikeEN-fs:/ >> rmdir src
Error: dir is not empty!
UnikeEN-fs:/ >> cd lib
UnikeEN-fs:/lib >> mkdir stdcpp
UnikeEN-fs:/lib >> cd stdcpp
UnikeEN-fs:/lib/stdcpp >> e
Goodbye!
(base) unikeen@ubuntu:~/Code/CS2303/source/Project_3/step2$

```

(a) Support Required Commands

```
unikeen@ubuntu: ~/Code/CS2303/source/Project_3/step2
```

File Edit View Search Terminal Help

(base) unikeen@ubuntu:~/Code/CS2303/source/Project_3/step2\$./fs
UNIkEEN-fs: >> gl 0
This is a file inode

- index: 0
- mode: 0
- link_count: 0
- size: 0
- timestamp: 0
- parent: 0
- direct: {0,0,0,0,0,0,}
- single_indirect: 0

UNIkEEN-fs: >> f
UNIkEEN-fs:/ >> mkdir ddd
UNIkEEN-fs:/ >> gl 0
This is a dir inode

- index: 0
- mode: 1
- link_count: 1
- size: invalid
- timestamp: 1685073718
- parent: invalid
- direct: {131,0,0,0,0,0,}

UNIkEEN-fs:/ >> gb 131

\$ represents '\0'

UNIkEEN-fs:/ >>

(b) Commands for debugging (**gi** shows information about inode, **gb** shows block content)

Fig. 8. Screenshots of Step 2

3 Step 3: Work Together

3.1 Implementation

3.1.1 Socket Programming

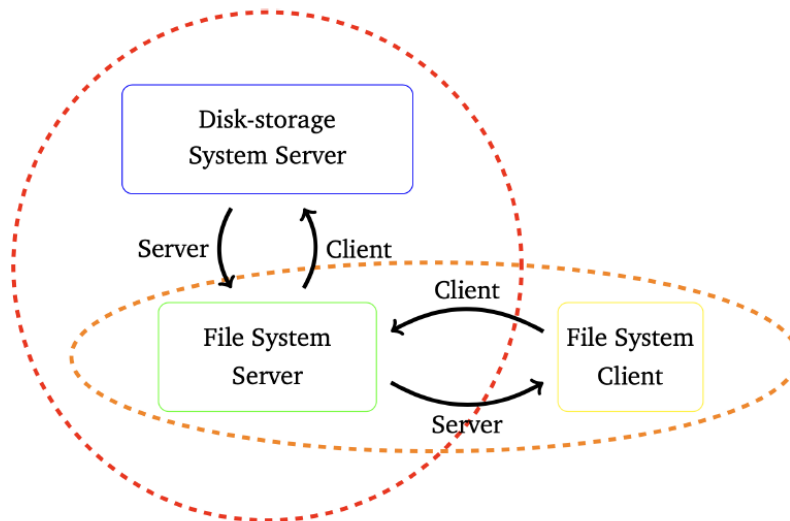


Fig. 9. Network File System Architecture

As shown in Fig. 9, in step 3, the file system has two sets of sockets, one connecting the simulated disk with the file system and the other connecting the file system with the client. When used with a single client, these two sets of sockets[2] in the system share many similarities in terms of implementation.

Connect and Listen The following is an example code snippet for client connection, using the `client.c` file as a reference:

```
struct sockaddr_in server_addr;
int client_socket;
client_socket = socket(AF_INET, SOCK_STREAM, 0);
...//Error handling
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(port);
...//Error handling
printf("Connected to server\n");
while (1) {
    ...//send command and recv response
}
```

```
close(client_socket);  
return 0;
```

After establishing the connection, the client can sequentially execute the `send()` and `recv()` functions in a loop to send commands to the server and receive instructions from the server.

Here is the core code for the server-side, using the code from `disk.c` as an example. In the updated `main()` function, it will now be used to continuously accept client connections. Upon a successful connection, it will enter the second level of the `handle_client()` function's loop to receive and process commands.

This design only supports one client successfully connected at a time. Once the client disconnects, the server continues listening for new client connection requests.

```
// Handle requests from the FS client  
void handle_client(SimDisk *disk, int client_socket) {  
    ... // Command categorization and execution, refer to sections 1.1.3 and 2.2.5  
    while (1) {  
        ... // Receive command and send response  
    }  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    struct sockaddr_in server_addr, client_addr;  
    socklen_t client_len;  
    int client_socket;  
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);  
    ... // Error handling  
    memset(&server_addr, 0, sizeof(server_addr));  
    server_addr.sin_family = AF_INET;  
    server_addr.sin_addr.s_addr = INADDR_ANY;  
    server_addr.sin_port = htons(port);  
    ... // Error handling  
    printf("Disk server started. Listening on port %d\n", port);  
    while (1) {  
        client_len = sizeof(client_addr);  
        client_socket = accept(server_socket, (struct sockaddr *)&client_addr,  
                               &client_len);  
        ... // Error handling  
        printf("New FS client connected. Client socket: %d\n", client_socket);
```



```

        handle_client(&disk, client_socket); // Successfully connected to the
            client, start handling commands
        printf("FS client disconnected. Client socket: %d\n", client_socket);
        close(client_socket);
    }
    close(server_socket);
    return 0;
}

```

Online Disk Operation In step 2, I have created a simulated disk where data is stored in memory in the form of blocks. In the third step, for the file system component, in theory, I only need to modify the read and write block functions in `diskop.c`. By establishing a socket connection to the simulated disk stored using `mmap` in step 1, I can execute the instructions from step 1 in the desired format.

Additionally, I have made modifications to the disk command format in step 1. It now receives block numbers and calculates the corresponding cylinder and sector numbers, instead of directly receiving cylinder and sector numbers.

As an example, here is the core code for reading a block, which is part of step 2:

```
memcpy(buf, disk_file[block_no], BLOCK_SIZE);
```

Now, the code has been modified as follows (error handling parts have been omitted):

```

typedef struct input_cmd{
    char type;
    int block_no;
    char info[BLOCK_SIZE];
} input_cmd;

input_cmd disk_command;
disk_command.type = 'R';
disk_command.block_no = block_no;
send(disk_server_socket, &disk_command, sizeof(disk_command), 0);
memset(buf, 0, BLOCK_SIZE);
recv(disk_server_socket, buf, BLOCK_SIZE, 0);

```

The `input_cmd` structure is defined for the convenience of data transmission, as the data may not be in the form of standard strings. Concatenating command strings can be challenging for the server-side processing, as discussed in the “A Key Point in Data Transmission” section.

Response Display In the localized file system implementation of step 2, the response to various commands can be directly displayed on the screen using `printf()`. However, in step 3, to ensure that command responses are displayed on the client-side, I have adopted an approach:

Prior to executing a command on the server-side, a `response` string is cleared firstly. I have implemented a new function called `_printf()` that can directly replace the `printf()` function used in step 2. Its purpose is to append the string to the end of the `response` string. After executing the command, the whole `response` string is sent to the client and displayed accordingly.

```
void _printf(const char* format, ...) {
    va_list args; va_start(args, format);
    int current_length = strlen(response);
    // format and append
    vsnprintf(response + current_length, RESPONSE_LENGTH - current_length,
               format, args);
    va_end(args);
}
```

To support formatted strings, I have followed the implementation of standard library functions like `printf()` and used a variable argument list (`va_list`). This allows the function to accept a variable number of arguments, similar to how `printf()` works.

Key Points in Data Transmission In step 3, this particular step can be prone to errors.

When performing operations such as reading and writing blocks, it is common to convert structures (e.g. `superblock`, `dir_item`) to character arrays for data transfer between the file system and disk. In such cases, it is crucial to use `memcpy()` instead of `strcpy()` for copying data. Additionally, when using the `send()` function, it is also crucial to use predefined constants or `sizeof()` for indicating the number of bytes to send, rather than relying on `strlen()`. Otherwise, if there are `null('\0')` characters within the data, it can result in incomplete copying and transmission.

During the process of transmitting command execution results between the file system and the client, the data being transmitted is in the form of standard strings. In this case, it is appropriate to use `strlen()` as the parameter for indicating the number of bytes in the `send()` function. However, it is important to note that the result of the command execution, i.e., the transmitted data, may be empty. In such cases, using `strlen()` can lead to a deadlock where the client continues trying to receive results without sending new commands, while the server is waiting to receive the next command. To handle this specific scenario, we can use `sizeof()` instead.

```
if (!strlen(response)){
    if (send(client_socket, response, sizeof(response), 0) < 0)
        perror("Failed to send response to client");
} else{
```

```
    if (send(client_socket, response, strlen(response), 0) < 0)
        perror("Failed to send response to client");
}
```

Furthermore, the client I implemented is similar to a shell. After receiving the command results from the file system, it continues to send the current path to be displayed on the client. The proximity of the two send calls may lead to the issue of socket “stickiness” or packet bundling. To address this problem, I have introduced some delays using `sleep()` between the two send calls to avoid packet bundling.

3.1.2 Persistence and Retrieval of File System

Magic Number In Linux, there are magic numbers used to identify file system types. For example, the magic number for EXT2/3/4 file systems is `0xEF53`[1]. In step 3, I have added a new field to the superblock called `s_magic`, which represents the magic number for my file system. Upon initialization, the magic number is set to `0xE986` (986 being the last two digits of my student ID). Every time the file system is started, the contents of the first three blocks on the disk are read and attempted to be parsed as the `superblock` structure. If the magic number matches, it indicates that a formatted file system already exists on the simulated disk, and the subsequent blocks containing the inode table are loaded into memory. Otherwise, the system waits for the user to input a formatting command.

As mentioned earlier, the superblock and inode table are loaded from the disk only once during startup, and subsequent reads will access the data in memory. Any modifications will be made to the data in memory and synchronized back to the disk.

Adaptation of Disk Size To simplify the development process, I have fixed the maximum number of blocks supported by the file system to 4096 and the maximum number of inodes to 1024 (same as in step 2). The first 131 blocks are reserved for the superblock and inode map. During file system initialization, we use the ‘I’ command implemented in step 1 to query and calculate the total number of sectors on the disk. If the number is less than or equal to 131, the file system cannot be initialized.

When loading an existing file system, we also query and calculate the total number of sectors on the disk and compare it with the information stored in the superblock. Therefore, the implemented file system requires a minimum of 132 sectors on the connected disk. If the number of sectors exceeds 4096, the file system will only utilize the first 4096 sectors.

```
int init_spb(){
    spb.s_magic = 0xE986; //set magic number
    int num_disk_block = get_disk_info(); //check disk size
    if (num_disk_block > MAX_BLOCK_COUNT) num_disk_block = MAX_BLOCK_COUNT;
    if (num_disk_block <= RESERVE_BLOCK) return -1;
```

```
...
}

int get_disk_info(){
    input_cmd disk_command;
    disk_command.type = 'I';
    send(disk_server_socket, &disk_command, sizeof(disk_command), 0);
    ...//Error handling
    int ret = recv(disk_server_socket, tmp, BLOCK_SIZE, 0);
    ...//Error handling
    int num1, num2;
    sscanf(tmp, "%d %d", &num1, &num2);
    return num1*num2;
}
```

3.1.3 Log Detailization

In steps 1 and 2, the required log information was quite rudimentary, consisting only of "yes" and "no" responses. Therefore, I have standardized the format of the log file, and the resulting example is as follows:

In disk.log:

```
[2023-05-26 11:48:59] [INFO] Received disk command:: I 0
[2023-05-26 11:48:59] [INFO] I commands executed successfully.
[2023-05-26 11:48:59] [DEBUG] Disk info: 1 cylinders, 1300 sectors per cylinder.
...
```

In fs.log:

```
[INFO] Received: cat b.cpp
[DEBUG] cat response:33a44bcd
[INFO] Received: i b.cpp 7 GG
[ERROR] 'i' command error: Illegal file name!
...
```

3.2 Running Result

Here are some screenshots of the runtime, please refer to the video for a detailed demonstration.

The screenshot displays four terminal windows and a log file. The top-left terminal shows the server's response to client commands, including directory listings and file operations. The top-right terminal shows the client's perspective, including the directory listing and file operation results. The bottom-left terminal shows the client's connection and disconnection. The bottom-right window shows the 'disk.log' file, which contains a detailed log of all server and client interactions, including timestamps and command details.

Fig. 10. Screenshot of Step 3

4 Further Development and Prospect

In this section, I will attempt to further develop and optimize the functionality based on step 3. It is important to note that not all code implementations discussed in this section are included in the submitted code.

4.1 Multi-Client Support

First, The code in the `main()` function of the file system server needs to be modified to create a new thread executing the `handle_client()` function whenever a client connects. The following is the modified code snippet:

```
client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
    &client_len);
...
printf("New client connected. Client socket: %d\n", client_socket);
pthread_t client_thread;
int* new_sock = malloc(sizeof(int));
*new_sock = client_socket;
if(pthread_create(&client_thread, NULL, handle_client, (void*) new_sock) < 0) {
    perror("Failed to create thread"); continue;}
}
```

Furthermore, in order to allow multiple clients to operate on different directories within the same server file system simultaneously, we need to modify the client-side code to keep track of the inode number of the current accessed directory, rather than maintaining it on the server side. This inode number should be sent along with the commands to the server.

Additionally, we need to address the issue of **thread synchronization**. A simple implementation involves maintaining a command queue within the file system. The commands are executed in a first-come, first-served (**FCFS**) manner, **with only one command being executed at a time**. This queue can be implemented using a linked list, and it should also store the directory inode number passed from the client.

4.2 Multi-User Support and Authority Management

A file system should be capable of providing different permissions for different users. The multi-user support implemented in the following approach is relatively simplistic. We can use a `uint8_t` variable to represent users (supporting up to 256 users).

In the `inode` structure of both step 2 and 3, we have reserved two bytes of space (`uint16_t reserve`). We can now repurpose this space as `uint8_t owner` and `uint8_t auth`. The former records the ID of the owner, who is the user that created the file and has read and write permissions. The latter is read as binary bits, with the first two bits indicating the read and write permissions for other users (0 means permission is granted, 1 means it is not). Before executing relevant commands, user IDs can be checked to enforce the appropriate permissions.

4.3 Directory Cache and Write-back Strategy

In the implementation of step 2 and 3, various operations such as reading, writing, creating, or deleting files involve accessing the inode of the current directory and iterating through its `dir_items` to find the desired entry. To optimize this process, we can introduce a caching mechanism. When a directory is first loaded, its disk blocks can be read and its 64 `dir_items` can be stored in an in-memory array. Subsequent accesses to the directory can then directly access the in-memory array. When the directory's inode is modified to allocate new blocks or is to be evicted from the cache, the changes will be written back to the disk.

The write-back strategy will also impact performance. In the implementation of step 2 and step 3, whenever there is a change in the inode table or superblock (such as inode allocation or deallocation), we immediately synchronize the modifications by writing them back to the disk. Additionally, the write-back operation **can be performed altogether during system shutdown** instead of synchronously updating on every modification, which would improve performance. However, it is important to note that this approach may lead to data loss in the event of an abnormal shutdown or power failure. Therefore, it becomes **a trade-off between performance and data safety**, and the actual implementation should consider both aspects.

5 Conclusion

In this project, we have gradually built a network file storage system:

- **Technology:** We used memory-file mapping to establish a simulated disk and designed the superblock and inode based on the ext2 file system in Linux to store file system metadata and files. Additionally, we utilized socket technology to enable communication between different parts.
- **Performance:** We achieved good performance in allocation and deallocation by utilizing the superblock and bitmap to track the availability of blocks and inodes. Furthermore, we introduced the concept of directory cache in subsequent optimizations, which further improved performance.
- **Scalability:** We modularized the instructions in both the disk and file system, making it easy to extend and develop new functionalities.
- **User Interface:** The client provides clear return value display for user-input commands, showing the current working directory in a highlighted color to emphasize the context. Additionally, detailed logging is implemented to record the results of all operations.

Through the development process, I have gained a deeper understanding of the principles behind technologies such as inodes and familiarized myself with C language development, particularly in the areas of string manipulation, memory management, and socket programming. This experience has allowed me to enhance my technical skills and expand my knowledge in these domains.

References

- [1] Linux Kernel, *The ext2, ext3 and ext4 File Systems*, 2023. Accessed: 2023-05-26.
- [2] SanjayRV, "A beginners guide to socket programming in c," 2021.