Project 3 Report
Name: Ciel Student id: 7

Contents

1 Introduction

1.1 Project Background and Objectives

Modern operating systems rely heavily on file systems to organize, store, and manage persistent data. At the heart of these file systems is the abstraction of secondary storage devices, which transform raw disk hardware into user-friendly interfaces. This project simulates this transformation process by requiring students to implement a virtual file system from the ground up, incorporating both a low-level disk subsystem and a higher-level file system interface.

The project is inspired by real-world systems and challenges, combining aspects of systems programming, network communication, data structures, and storage management. Provides a comprehensive learning experience by emulating the internal workings of the storage and file management subsystems of an operating system.

This project is divided into multiple phases with the following main objectives:

- Implement a disk-like secondary storage server as an Internet-domain socket server.
- Construct a simple yet functional file system on top of the disk server.
- Extend the file system to a separate server using UNIX-domain sockets.
- Implement user management within the file system to support isolated environments for different users.
- Additional challenges include designing a block cache for performance optimization, supporting concurrent multi-client operations with synchronization mechanisms, and ensuring crash consistency using journaling techniques.

1.2 Key Technologies Involved

Our key technical details include: Inode-based structure, free place management, TCP socket communication, and LRU block caching.

1.3 Report Overview

First, we began by outlining the required functionalities and key features of our virtual disk and virtual file systems. **Subsequently**, we delved into the implementation details of both systems, focusing on the challenges encountered during development and the corresponding optimization strategies. **Then**, building upon the foundational file system, we introduced a caching mechanism and analyzed its performance. **Finally**, we concluded the project with a brief summary of our work.

2 Detailed Requirements

2.1 Basic Disk-Storage System

The basic storage system in this project implements a simulated disk as a socket-based server, emulating the behavior of the physical disk while allowing remote client interaction. It is designed to operate based on a geometric layout defined by cylinders and sectors, with a fixed block size of 512 bytes per sector. Upon initialization, the disk server requires the name of a backing file, the number of cylinders and sectors per cylinder, a track-to-track seek delay (in microseconds) and a TCP port number to listen for client connections. Internally, the server memory-maps the backing file to

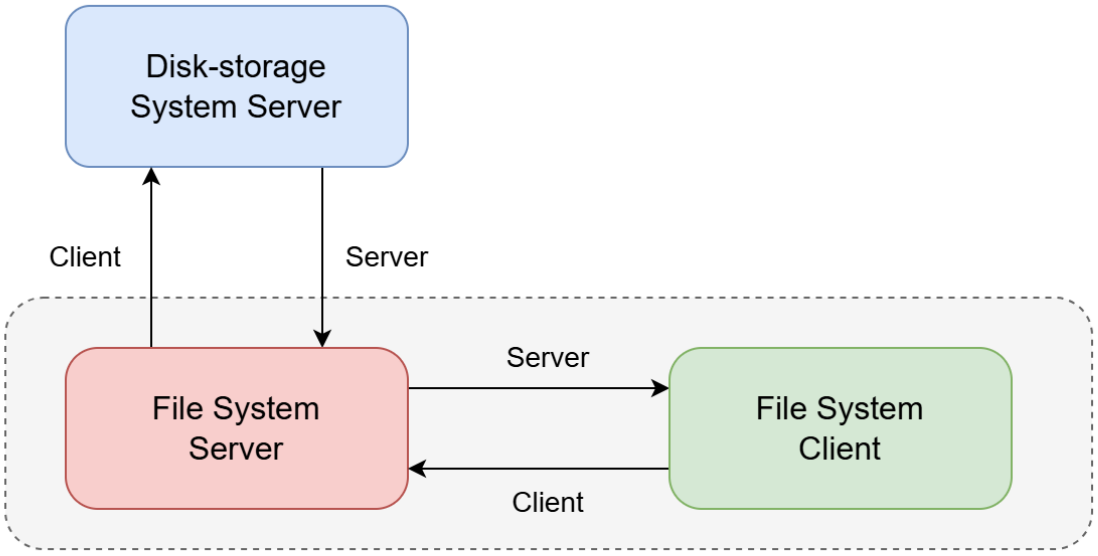

Figure 1: File system architecture

emulate a persistent storage medium and calculates the total size of the simulated disk accordingly. Seek time is simulated by introducing delays proportional to the difference between the current and target cylinder positions, effectively modeling realistic disk behavior.

The server supports several core commands: an information command (`I`) to retrieve the geometry of the disk, a read command (`R c s`) to fetch raw 512-byte content from a specified sector, and a hexadecimal read command (`X c s`) to return the same content in a readable hexadecimal format for debugging purposes. For writing, the `W c s l data` command enables storing up to 512 bytes of user-provided data in a given sector. If the data length is less than 512, the remaining bytes in the sector are zero-padded to maintain consistency. Each operation rigorously checks for parameter validity, ensuring that read- and write-requests fall within the bounds of the disk geometry. When errors such as invalid cylinder or sector indices occur, or if data lengths are incorrect, the server responds with a standardized error message (`No`) and logs the event.

All client-server communication is conducted via TCP sockets. The client operates in an interactive loop, enabling users to enter disk commands, transmit them to the server, and display the results. It is also designed to handle binary data and non-printable characters gracefully, particularly for debugging output, where hexadecimal representation becomes essential. The parsing mechanism distinguishes between command tokens and data accurately, accounting for complexities such as embedded whitespace or invisible characters in the data field.

2.2 Basic File System

The basic file system builds on top of the virtual disk server and implements an inode-based file system that supports standard file and directory operations. It provides a higher-level abstraction for persistent data management while internally managing disk blocks, inodes, and directories through structured mechanisms. Upon initialization, the file system must format the disk and configure essential metadata structures, including a superblock, free block bitmap, and inode tables. These are critical for defining the physical layout and logical structure of the file system.

The file system operates with a UNIX-domain socket interface and supports a command-line client. It handles a set of structured commands, including file creation, deletion, read, write, insert, and delete operations. Directory management is realized using a special type of inode that stores directory entries. Commands such as `mkdir`, `rmdir`, `cd`, and `ls` enable creation, deletion, navigation, and listing of directories. The current working directory is tracked and updated accordingly, with full support for relative and absolute paths, as well as special tokens such as `.` and `..`.

To manage users, the system incorporates a lightweight multi-user model. Users are identified by numeric UIDs, and upon logging in, a user-specific home directory is created if it does not already exist. UID-based ownership and permission bits are associated with inodes. Only UID 1 is authorized to format the file system using the command `f`.

On the one hand, the file system functions as a server that receives requests from clients and performs corresponding file operations. On the other hand, to ensure long-term data persistence, the file system must involve a disk system rather than relying solely on memory storage. In this case, the file system acts as a client of the disk system, maintaining data non-volatility by storing data on the disk. The structure diagram of this project is illustrated in Fig. 1.

3 Basic Disk-Storage System

3.1 Extra Reading Operation

The standard read operation `R <ncyn> <nsec>` can successfully retrieve and display the text stored in a sector. However, in practice, not all files within a file system are human-readable text documents. Many files may contain non-printable ASCII characters or characters that, even if visible, carry no meaningful interpretation. Moreover, during the development process, it is often necessary to inspect the raw contents of disk storage for debugging purposes. In such cases, it is more useful to view the raw binary data rather than its encoded textual representation. To address this need, as shown in Lst 1, we have introduced a new command that outputs sector data in hexadecimal format, enabling more flexible and comprehensive read operations.

```c
int handle_rx(tcp_buffer *wb, char *args, int len) {
    int cyl;
    int sec;
    char buf[512];

    if (sscanf(args, "%d %d", &cyl, &sec) != 2) {
        Log("Invalid command format for READ: %s", args);
        reply_with_no(wb, NULL, 0);
        return 0;
    }

    if (cmd_r(cyl, sec, buf) == 0) {
        char response[512 * 2 + 1];
        char *ptr = response;

        for (int i = 0; i < 512; i++) {
            ptr += sprintf(ptr, "%02x", (unsigned char)buf[i]);
        }
        reply_with_yes(wb, response, strlen(response));
    } else {
        reply_with_no(wb, NULL, 0);
    }
    return 0;
}
```

Lst 1: Disk API for hexadecimal format reading

As shown in Lst. 1, what we have done is to add a new command handling function `handle_rx` in ***server.c***. This function, like the original function `handle_r`, uses the `cmd_r` disk API to read data from the disk. The key difference lies in how the retrieved data are returned to the client. While `handle_r` returns the data directly as a string, `handle_rx` first converts the raw data into a hexadecimal representation before returning it.

The core logic is as follows: `sprintf(ptr, "%02x", (unsigned char)buf[i])`. Each raw byte is explicitly cast to an unsigned character to prevent it from being misinterpreted as a negative number during formatting with `sprintf`. For example, if `char buf[i] = -1`, failing to cast it would result in `%x` outputting `ffffffff`, which after truncation becomes `ff`.

Since one byte consists of 8 bits and a single hexadecimal digit represents 4 bits, each byte is represented by two hexadecimal digits. The format specifier `%02x` ensures that each byte is displayed as a two-character hexadecimal value, padding with leading zeros if necessary. This strict formatting approach effectively prevents unexpected formatting errors during conversion. Consequently, we can confidently set the final output buffer size at 1025 bytes (512 bytes $\times$ 2 + 1 for the null terminator) without risking troublesome segmentation faults.

3.2 Parameter Parsing

Robustness in the command parsing module is essential to ensure the correctness and security of the system. As shown in Lst. 1, the command-handling functions (the `handle_*` series) use `sscanf` to parse two integer parameters (*ncyl* and *nsec*) from the input of the client, instead of the more traditional **atoi**. This design decision reflects careful consideration of both security and input validation.

The `atoi` function returns 0 when parsing fails due to an invalid string (e.g., “abc”). However, in the context of disk addressing, 0 is a valid cylinder or sector index. As a result, using `atoi` would make it impossible for the server to distinguish between a valid request to access cylinder 0, sector 0 and an invalid, non-numeric input. This ambiguity may lead the system to treat malformed commands as legitimate operations, posing a risk to data consistency.

Unlike `atoi`, `sscanf` provides a mechanism to detect parsing failures by returning the number of successfully matched and assigned fields. The following check can accurately detect formatting errors: `if (sscanf(args, "%d %d", &cyl, &sec) != 2)`. This ensures that only commands strictly matching the format `R <ncyl> <nsec>` are processed further, effectively preventing the execution of malformed commands and enhancing system stability from the ground up.

```
int cmd_w(int cyl, int sec, int len, char *data) {
    if (cyl >= disk._ncyl || sec >= disk._nsec || cyl < 0 || sec < 0) {
        Log("Invalid cylinder or sector: cyl=%d, sec=%d", cyl, sec);
        return 1;
    }
    if (len > BLOCKSIZE || len <= 0) {
        Log("Invalid data length: %d (must be 1-512)", len);
        return 1;
    }

    off_t offset = BLOCKSIZE * (cyl * disk._nsec + sec);
    usleep(1000 * abs(cyl-cur_cyl) * disk.ttd);
    cur_cyl = cyl;

    if (offset + BLOCKSIZE > disk.FILESIZE) {
        Log("Offset out of bound: offset=%ld, file size=%ld", offset, disk.FILESIZE
    );
        return 1;
    }

    memcpy(&disk.diskfile[offset], data, len);
    if (len < BLOCKSIZE) {
        memset(&disk.diskfile[offset + len], 0, BLOCKSIZE - len);
    }
    Log("Wrote sector: cyl:%d, sec=%d, len=%d", cyl, sec, len);
    return 0;
}
```

Lst 2: Disk API for writing data to a block

3.3 Data Validity Check and Zero-Filling Mechanism

In this project, we simulate a disk system with a fixed block size of 512 bytes. Consequently, as shown in Lst. 2, during the implementation of the `cmd_w` write operation function, we perform strict validity checks on the length of the input data. Additionally, we apply zero-padding when the data length is less than 512 bytes to ensure system stability and data consistency.

3.3.1 Enforcement of Data Length Constraints

The write operation is restricted to a data length that satisfies the condition: $1 \leq \text{len} \leq 512$. This constraint is imposed for the following reasons:

Preventing Out-of-Bounds Writes: Each write operation is limited to a single sector. If the data exceeds 512 bytes, it may overwrite adjacent sectors, corrupting other stored data, and severely compromising file system consistency.

Avoiding Invalid or Illegal Operations: Writing zero or negative bytes is non-sensical and may cause unexpected behavior in functions like memcpy, potentially leading to system crashes.

3.3.2 Zero Padding for Partial Blocks

When length of the written data is less than a full block, the remaining bytes are filled with zeros using: `memset(&disk.diskfile[offset + len], 0, BLOCKSIZE - len)`.

Without zero-padding, the residual portion of the sector may retain stale data from previous writes, leading to ambiguity and inconsistencies during subsequent reads.

Regardless of the actual data length, the read operation always retrieves the full 512 bytes. Although inserting a single null terminator after valid data may suffice when reading strings, this approach falls short in practice since disk contents include not only text, but also structured data (e.g., inode blocks). Therefore, partial padding cannot guarantee read-write consistency.

Based on these considerations, we adopt a comprehensive zero-filling strategy for the remaining space in a block. This method offers an additional benefit: when resetting a block, writing a single zero byte triggers the system to automatically zero out the rest, simplifying block management.

4 Basic File System

4.1 Structure Size Alignment

In this project, the `dinode` structure serves as the core metadata representation for each file or directory in the file system. To improve storage efficiency and simplify disk read/write operations, the `dinode` size is required to be a divisor of the block size of `BSIZE = 512` bytes.

During development, we observed the following behavior: `uint` occupies 4 bytes, and `ushort` occupies 2 bytes. When an odd number of `ushort` fields is declared in a structure, the compiler automatically adds 2 bytes of padding, making the overall structure size a multiple of 4. Upon further investigation, we found that this behavior results from C language structures being aligned in memory according to platform-specific ABI (Application Binary Interface) rules. The key alignment principles are as follows:

- Each member's address must be aligned with a multiple of its type size.
- The total size of the structure must be aligned with the alignment requirement of its largest member.
- The compiler may insert padding bytes between members or at the end of the structure to satisfy these alignment constraints.

In our case, the largest member type in `dinode` is a 4-byte unsigned integer (`uint`). If an odd number of `ushort` members precede a `uint` , the compiler inserts 2 bytes of padding to align the subsequent `uint` to a 4-byte boundary.

In some cases, we may explicitly add padding, such as `char pad[4]` , to ensure that the structure meets specific alignment or size requirements. However, care must be taken to account for the compiler's automatic alignment behavior; otherwise, manual padding could unintentionally cause the structure size to misalign with the block size.

Because the file system imposes strict requirements on low-level storage structures, such as `dinode` , any misalignment can lead to unpredictable behavior and significantly increase debugging complexity. To mitigate such risks, we use a compile-time assertion:

```
static_assert(BSIZE % sizeof(dinode) == 0, "alignment fault")
```

This ensures that the size of `dinode` is a divisor of the block size. If the condition is satisfied, the compilation proceeds normally. Otherwise, the build fails with a clear error message, preventing potential runtime issues due to misalignment.

```
typedef struct {
    char name[MAXNAME];
    short type;
    uint inum;
    uint size;
    uint mtime;
    uint ctime;
    uint owner;
    ushort perm;
} entry;
```

Lst 3: Definition of entry structure

4.2 Directory Organization

In the design of the virtual file system in this project, directories are the core components that enable hierarchical file management. Similarly to real-world file systems, a directory is essentially **a special type of file** whose data consists of a set of directory entries representing the contents of that directory.

As shown in Lst. 3, we define the directory entry using a structure named `entry` . The contents of a directory are stored as a sequential collection of entry structures. Since each entry occupies a fixed number of bytes, the directory can be traversed directly using offset-based access. Reading a directory involves retrieving all data blocks associated with its inode and parsing them as an array of entry structures.

It is important to note that in our design, **the size of `entry` does not align with the block size**. As a result, a single entry may span across two disk blocks. This design choice complicates the process of loading an entire directory block into memory and then indexing all directory entries within it. Such an approach becomes significantly more complex and error-prone under these circumstances.

To address this problem, we adopt a method of reading directory entries directly from disk using a combination of `base address + offset + data length`, as shown in Lst. 4. This strategy allows the system to implicitly handle the translation from virtual to physical addresses, without requiring knowledge of the exact memory alignment of directory entries. Consequently, we avoid the challenges posed by misaligned structures.

However, this approach comes with a trade-off: it incurs frequent disk access operations, resulting in substantial overhead. Because our project simulates only disk seek time and directory accesses often occur within the same cylinder, this simplified model does not reflect the full cost. But in real systems, even transferring data from disk to memory represents a significant overhead. This is indeed a limitation of our design. Fortunately, the caching mechanism that we implemented later in 5 effectively mitigates this issue, reducing the performance penalty of repeated disk accesses.

```c
int dir_lookup(inode *dp, const char *name, uint *inum_out) {
    entry e;
    for (uint off = 0; off + sizeof(e) <= dp->size; off += sizeof(e)) {
        readi(dp, (uchar *)&e, off, sizeof(e));
        if (strncmp(e.name, name, MAXNAME) == 0) {
            if (inum_out) *inum_out = e.inum;
            return e.type;
        }
    }
    return 0;
}
```

Lst 4: Implementation of directory looking up

```c
struct superblock {
    uint magic;             /* magic number */
    uint size;              /* total block number of disk */
    uint bmapstart;         /* starting block number of bit-map */
    uint datastart;         /* starting block number of data zone */
    uint ninodeblock;       /* number of inode blocks */
    uint inodeblock[123];   /* inode block index */
};
```

Lst 5: Superblock structure

4.3 File System Organization

The superblock is stored in the first sector of the disk (i.e., `ncyl = 0`, `nsec = 0`) in our file system. As shown in Lst 5, it contains the fundamental metadata and reveals the organization of our file system. In the following sections, we outline the rationale behind the design of each attribute in the superblock and explain their respective functions.

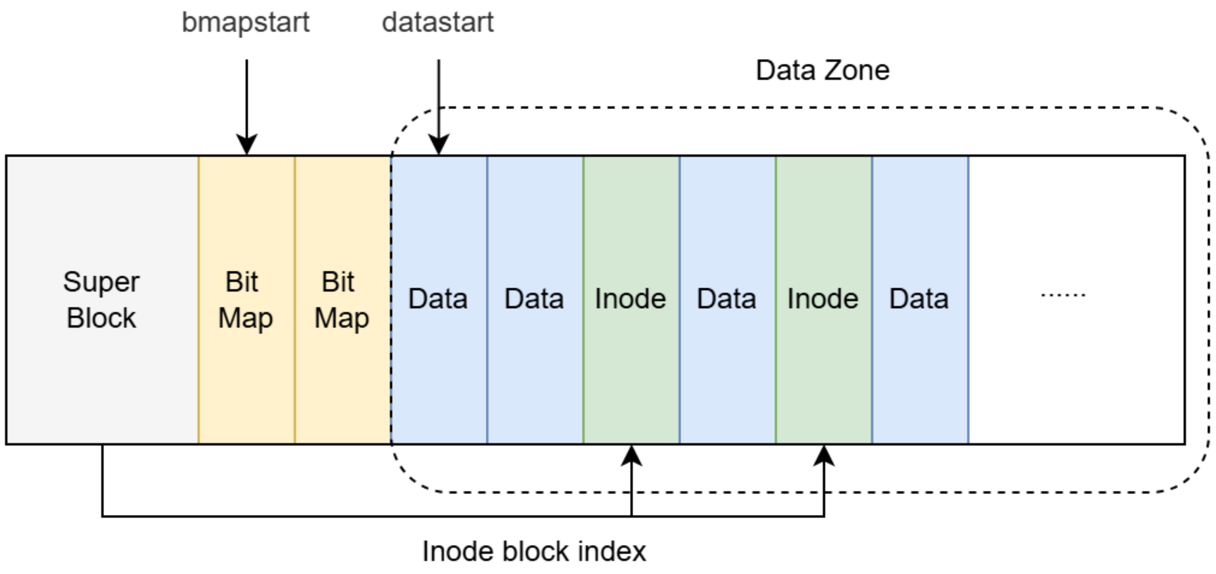

Figure 2: File system organization

4.3.1 Dynamic Allocation of Bitmap Blocks

The file system adopts a bitmap mechanism to manage free blocks. Unlike traditional implementations that hardcode the location of bitmap blocks, our system dynamically calculates the number of bitmap blocks required based on the total disk size. As illustrated in Fig. 2, these bitmap blocks are stored in the low address, after the superblock and before the data zone. The superblock records both the starting block number of the bitmap (`bmapstart`) and the starting block number of the data region (`datastart`). The blocks between these two markers are allocated for the bitmap.

This design improves the adaptability and scalability of the file system, enabling it to support disks of varying sizes without requiring structural modifications.

4.3.2 Dynamic Allocation of Inode Blocks

Unlike conventional file systems that reserve a fixed region at the beginning of the disk for inode storage (commonly referred to as the inode table), our system stores inodes dynamically within the data region, treating them the same as regular files or directories, as shown in Fig. 2. Inodes are allocated as needed, rather than occupying pre-reserved space.

This approach simplifies the overall layout of the data and improves space utilization efficiency. To support this mechanism, the superblock maintains the number of inode blocks (`ninodeblock`) as well as an array of their block numbers (`inodeblock[123]`), allowing the system to locate all inode blocks effectively.

4.3.3 System Limitations

Superblock Size Constraint In this system, the superblock is restricted to a single block (512 bytes). Since it must store the locations of all inode blocks, the number of supported inodes is limited by the size of the array, up to a maximum of 123 inode blocks. This constraint may become a bottleneck in large-scale file systems. In future expansions, we may allow the superblock to span multiple blocks to accommodate broader use cases. This is also why we store the starting block number of the bitmap in the superblock, rather than fixing it at block 2.

Non-reusable Inode Blocks Once a block is allocated as an inode block, it is not reclaimed or reused, even if its contents are cleared. This design is due to the fixed assignment of inode numbers (`inum`) in ascending order when creating directory entries. To access a directory entry, as shown in Lst 6, the system performs a linear search through the inode blocks to locate the corresponding inode. Reducing the number of inode blocks and altering the indexing would require broadcasting updates to all affected entries to revise their inum values, a process that is complex, error-prone, and incurs significant overhead.

Therefore, we adopt a compromise approach: dynamic allocation without dynamic reclamation. This is not an inherently poor strategy. Although a cleared inode block is not actively reclaimed, it may still be reused in future inode allocations.

```
#define IBLOCK(i) (sb.inodeblock[(i) / INODES_PER_BLOCK])  // calculate inode block
#define IOFFSET(i) ((i) % INODES_PER_BLOCK)             // calculate offset in block

// get the inode numbered inum
inode *iget(uint inum) {
    uchar buf[BSIZE];
    if (inum / INODES_PER_BLOCK >= sb.ninodeblock) {
        Warn("Invalid inode number");
        return NULL;
    }
    read_block(IBLOCK(inum), buf);

    dinode *dip = ((dinode *)buf) + IOFFSET(inum);
    if (dip->type == 0) return NULL;

    ...
}
```

Lst 6: Algorithm to locate inode numbered inum

4.4 Naming Constraints for Directory Entries

In the virtual file system designed for this project, **we explicitly prohibit the coexistence of files and directories of the same name within a single directory**. This is not merely a restrictive constraint, but a deliberate design choice based on architectural considerations, lookup efficiency, and operational simplicity.

Firstly, in our system, the directory entry structure is unified for both files and directories, meaning that they share a common namespace. During a lookup, each entry is uniquely identified by its name field, without distinguishing between file and directory types. Allowing duplicate names would necessitate additional type checking (i.e., comparing both `name` and `type`), increasing complexity in both implementation and maintenance.

Secondly, enforcing name uniqueness simplifies the permission management. To support permission control, we introduce a `chmod` operation in the form of `chmod <entry name> <permission>` . This command specifies only the name of the entry without including its type. Permitting name duplication would lead to ambiguity. Requiring users to specify an additional `<entry type>` would complicate operations and degrade the user experience.

Lastly, this naming policy aligns with the design of traditional Unix-like file systems, which also disallow files and directories with identical names within the same directory. This widely adopted

Permission	**File**	**Directory**
0	Not viewable	Not accessible
1	Viewable, non-editable	Accessible, entries non-editable
2	Editable	Editable

Table 1: File system permission machanism

and well-tested paradigm ensures compatibility and reliability.

In summary, adopting a unified namespace and excluding duplicate names among files and directories within a directory is a rational decision. Reflects a balanced trade-off between functional completeness, implementation simplicity, and user usability.

4.5 Permission Mechanism

To support multi-user access control, this file system implements a simplified permission mechanism. The primary objective of this mechanism is to regulate user access to files and directories, thus ensuring data confidentiality and system security.

4.5.1 Permission Field Definition

Each file or directory contains two permission-related fields in its inode structure:

- `perm` : A permission flag of type `ushort`, indicating the access level of the file or directory.
- `owner` : The owner's UID of type `uint`, specifying the file owner or directory owner.

We adopt a simplified permission model for the `perm` field, allowing only three possible values: 0, 1, and 2, as illustrated in Tab. 1. These permissions apply only to external users, those who are neither the owner nor the superuser. The superuser (UID = 1) and the file / directory owner have full access rights, regardless of the `perm` value.

4.5.2 Permission Verification Logic

All file operations that involve inodes, such as reading, writing, or deleting, must be preceded by permission checks on both the target file and the directory in which it resides. To perform these checks, we introduce a helper function `has_permission` defined as follows:

```
int has_permission(inode *ip, int required) {
    if (current_uid == 1 || current_uid == ip->owner) return 1;
    return ip->perm >= required;
}
```

Lst 7: Helper funtion: has_permission()

Here, `ip` refers to the inode of the target object and `required` denotes the permission level needed for the intended operation (e.g., reading requires level 1, while writing requires level 2). If the requesting user is the owner or the superuser, access is granted immediately. Otherwise, the system checks whether the file's permission level meets or exceeds the required threshold for the operation.

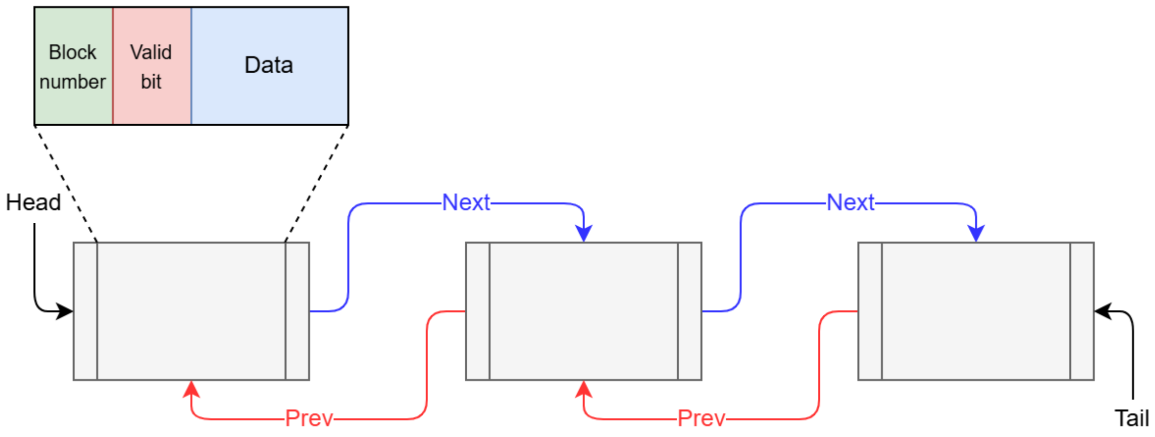

Figure 3: Structure of cache

4.5.3 Permission Management Command

By default, files or directories are created with permission level 1 (read-only). In practical use, users may need to modify the permissions of the files or directories they own. To support this functionality, we designed the int `cmd_chmod(char *name, int perm, int kernel)` function in the file system to implement the chmod command.

Notably, this function includes a parameter `kernel` that indicates whether the file system is operating in kernel mode. This design choice addresses a specific scenario: although the root directory is set to read-only, a new user must create a directory under the root upon login. At that moment, the file system's UID is that of the new user, who lacks the necessary permissions to modify the root directory. To resolve this, the system temporarily switches to kernel mode to bypass standard permission checks.

This design introduces potential security risks, since setting kernel to a non-zero value allows permission changes without restrictions. However, we still opted for this compromise to maintain the simplicity of our implementation.

4.5.4 Directory Permission Characteristics

Our directory permission system features an interesting mechanism: to modify a directory entry, it suffices to check the permissions of the entry itself and its immediate parent directory, there is no need to verify the permissions of higher-level directories. This design allows for scenarios where a parent directory is read-only, but its subdirectory is writable; in such cases, modifications within the subdirectory are still permitted. This design is motivated by three main considerations:

Complexity and Overhead Requiring permission checks across all ancestor directories during file access could lead to repeated disk accesses, resulting in significant overhead. Since file access is a fundamental operation in file systems, we aim to keep it as efficient and straightforward as possible by limiting permission checks to the immediate directory.

Permission Semantics In our operating system, modifying a writable subdirectory within a read-only parent does not alter the core attributes (such as `name` or `type`) of the parent directory entries. From this perspective, the behavior aligns with the intended logic of our permission mechanism.

Structural Constraints The root directory is designated as read-only for security reasons, to prevent unprivileged users from deleting or creating files and directories arbitrarily within it. If outer directory permissions were allowed to propagate and restrict inner ones, no directory would ever be writable, as the root is permanently set to read-only. This outcome would be unacceptable and overly restrictive for system functionality.

5 Cache

5.1 Design and Implementation

5.1.1 Cache Structure

Our caching mechanism is built upon the concept of cache entries, with the structure defined in Lst.8. The cache operates at the granularity of physical disk blocks, with each cache entry storing a block number, the corresponding block data, and a set of status flags. Additionally, each entry contains pointers to its preceding and succeeding entries.

These cache entries are linked by pointers to form a doubly linked list, which constitutes the core structure of our cache. To facilitate efficient list operations, we maintain two global variables, head and tail, which point to the beginning and end of the list, respectively. The structural diagram is presented in Fig.3.

```
typedef struct CacheEntry {
    int blockno;
    uchar data[BSIZE];
    int valid;
    struct CacheEntry *prev;
    struct CacheEntry *next;
} CacheEntry;
```

Lst 8: Cache entry structure

5.1.2 Cache Management

To support operations such as updating, searching, and clearing the cache list, we define several functions. Cache replacement is handled using the Least Recently Used (LRU) policy.

- `init_block_cache()` : Initializes the cache system.
- `find_in_cache(int blockno)` : Checks whether a specific block is present in the cache.
- `move_to_front(CacheEntry *entry)` : Moves a cache entry to the front of the list, marking it as recently used.
- `evict_and_insert(int blockno, uchar *data)` : If the cache is full, evicts the least recently used entry and inserts a new block.
- `clear_block_cache()` : Clears the entire cache.

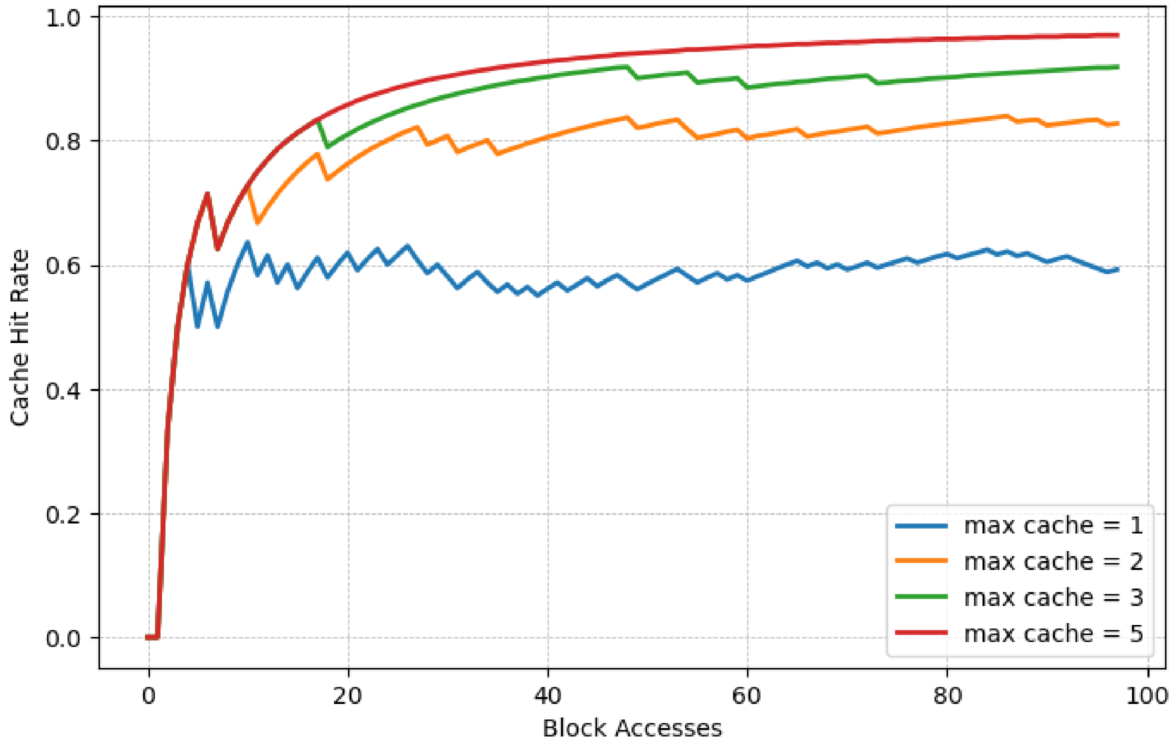

Figure 4: Cache hit rates under different cache sizes

5.1.3 Workflow

Before reading a block, the system first calls `find_in_cache()` to check if the block is already cached. If the block is found (cache hit), it is read directly from memory and `move_to_front()` is called to update its position in the LRU list. If the block is not found (cache miss), it is read from the disk and then cached in memory via `evict_and_insert()` . All write operations update both the cache and the corresponding disk copy to ensure consistency.

5.2 Performance Evaluation

Due to the instability and variability of direct runtime measurements, we instead evaluate cache performance by examining the cache hit rate. We conducted a series of identical file system operations while varying the cache size (`CACHE_CAPACITY`) across values of 1, 2, 3, and 5 (representing the maximum number of cache entries). The corresponding changes in cache hit rates recorded in the system log are shown in Fig.4.

Given that the operations we designed access a relatively narrow range of disk blocks, we observe that even a cache size of 5 allows disk requests to consistently hit in the cache, with hit rates approaching 100%. Remarkably, even when the cache holds only a single block, the hit rate remains as high as 60%.

One key reason for such high cache efficiency lies in our directory lookup algorithm, discussed in 4.2. This algorithm repeatedly accesses the same disk block during lookups, which would normally incur significant I/O overhead. However, with caching enabled, these repeated accesses translate into repeated hits on the same cache entry, thereby significantly boosting the cache hit rate.

6 Conclusion

In this virtual file system project, we successfully built a client-server-based file system. Throughout the development process, we learned to apply modular design principles and system-level debugging techniques, while also strengthening our grasp of key technologies such as mmap, network programming, and inode structure design. Despite encountering complex challenges such as data consistency and caching strategies, we achieved satisfactory results through iterative testing and optimization.

Lastly, **We express our heartfelt gratitude to Professor Shengzhong Liu and Teaching Assistant Wei Wang for their thoughtful guidance and patient support throughout the course of this project.** Their help served as a vital bridge between theoretical knowledge and practical implementation.