# Operating Systems Designs
# Project3 Report: Inode File System

## (2023-2024 Autumn Semester)

| | |
|---|---|
| **Name:** | Chuheng Du |
| **Student No.:** | 522031910786 |
| **Date:** | May 29, 2024 |

## Contents

# 1 Experiment Objective

## 1.1 Experiment overview

In this experiment, we hope to implement a simple file system[1] based on the inode structure. The file system should be able to support basic file and directory operations, such as creating, deleting, reading, and writing files, as well as creating and deleting directories. The file system should also be able to support the operations of listing files in a directory and changing the current directory.

Also, our file system should be able to support multi-user operations. We need to implement the user management module, which can create, delete, and modify users, and assign different permissions options to files and directories.

As our file system supports multi-client concurrent operations, we need to implement concurrency control mechanisms to ensure the correctness of the file system.

## 1.2 Experiment environment

The experiment is implemented in the Ubuntu 23.04 LTS operating system. The programming language used is C/C++. The experiment is implemented in the user space, and the file system is implemented as a user-level file system. The concurrency control mechanism is implemented using the unix semaphores.

## 1.3 Project architecture

The project is divided into the following modules:

1. **Basic Disk Server**: This module simulates the behaviour of the disk. It provides the basic read and write operations of the disk and exposes the disk interface to upper layers as a tcp/ip server.

[1]The source code of this project can be found at https://github.com/s7a9/toy-inode-filesystem, which will be made public after the deadline of the project.

2. **Block Manager**: This module manages the allocation and deallocation of disk blocks. It provides the interface for the upper layer to allocate, free, load, unreference and flush blocks. This module also acts as a memory pool for the upper layer to cache blocks.
3. **Inode File**: This module implements the operations on a single inode file. It provides the interface for the upper layer to create, delete, insert, read, write, and truncate files.
4. **Directory Tree**: This module implements the operations on a directory. It provides the interface for the upper layer to create and delete directories, as well as list files in a directory and change the current directory.
5. **User Information**: This module stores the user information in a special file named `userfile` under the root directory. The user information includes user id and user name.
6. **Working Directory**: This module provides a context for each connecting client, as well as manages locks.
7. **Bytepack**: This module provides as a utility to serialize and deserialize data structures.
8. **Network utils**: This module provides as a utility to establish tcp/ip server and serve requests, or connect to a tcp/ip server and send requests.

## 2 Utilities

### 2.1 Descriptions

The file system can be viewed as a three layers structure as shown in figure 1. All three parts communicate over tcp/ip protocol, so we choose to design a simple network protocol to communicate between the layers.

### 2.2 Bytepack

According to the homework requirements, all data transfered on tcp/ip sockets are ASCII strings. However, we need to transfer binary data structures between the layers. To solve this problem, we design a simple bytepack protocol to serialize and deserialize data structures. The bytepack protocol is designed to be similar to the C `printf` and `scanf` functions, which can serialize and deserialize data structures in a similar way.

```
// Example code for operation CATCH file
bytepack_t request, response;
bytepack_init(&request, 1024);
bytepack_init(&response, 1024);
std::string filename;
std::cin >> filename;
// Pack the request
```

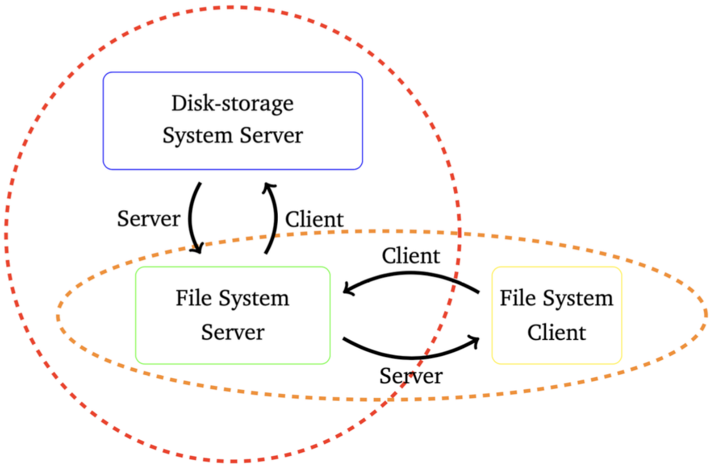

Figure 1: File System Structure

```
bytepack_pack(&request, "is", OP_CAT, filename.c_str());
bytepack_send(server_fd, &request);
bytepack_recv(server_fd, &response);
// Handle the response
bytepack_unpack(&response, "i", &result);
if (result != 0) { // Error
    std::cout << msg(result);
} else { // Success, create a buffer to hold the file content
    size_t size;
    bytepack_unpack(&response, "l", &size);
    char *data = new char[size];
    bytepack_unpack_bytes(&response, data, &size);
    std::cout.write(data, size);
    delete[] data;
}
```

### 2.3 Network Utils

The network utils module provides a simple interface to establish tcp/ip server and serve requests, or connect to a tcp/ip server and send requests. The interface is shown below:

```
int initialize_server_socket(int port);
int run_server(int sock_fd, void* (*handler)(void*));
```

```c
int connect_to_server(const char *ip, int port);
```

Whenever a client connects to the server, the server will create a new thread to handle the client request. The handler function should be implemented by the user and should be thread-safe.

## 3 Basic Disk Server

### 3.1 Descriptions

The basic disk server is implemented as a tcp/ip server and simulate a physical disk with fixed block size of 256 bytes. The disk structure is organized by cylinders and sectors. Additionally, the disk server records the time for heads to move from cylinder to cylinder.

#### 3.1.1 Protocol

- Information list: Returns the cylinder number and sector number of the disk.
- Read data: Read a whole block at a given cylinder and sector from disk.
- Write data: Write a whole block at a given cylinder and sector to disk.
- Exit: Close the connection and exit the server.

### 3.2 Design and Implementation

The major data structures used in the basic disk server are shown below:

```c
typedef struct {
    int num_cylinders;      // Number of cylinders
    int num_sectors;        // Number of sectors per cylinder
    int sector_move_time;   // Time to move between adjacent sectors
    char *diskfile;         // Pointer to the disk file buffer
    int current_cylinder;   // Current cylinder
    int current_sector;     // Current sector
    int total_time;         // Total time taken to serve requests
    sem_t mutex;            // Mutex to protect the disk structure
} disk_t;
```

#### 3.2.1 Map disk file to memory

The disk file is mapped to memory using the `mmap` system call. The disk file is opened in read-write mode and mapped to memory with the `PROT_READ | PROT_WRITE` flags. The disk file is then closed, and the disk structure is initialized.

To locate a block in the disk file, we can use the following formula:

$$\text{index} = \text{cylinder no.} \times \text{section no.} \times \text{block size}$$

Hence, the read and write operations can be implemented with the `memcpy` function.

#### 3.2.2 Simulations

In the project, we need to simulate the time taken to move the disk head between adjacent sectors. We use the `usleep` function to simulate the time taken to move the disk head. The time taken to move the disk head between adjacent sectors is stored in the `sector_move_time` field of the disk structure.

Whenever a read/write operation is performed, we wait for a certain time and then update the current cylinder and sector number. The total time taken to serve requests is stored in the `total_time` field of the disk structure.

#### 3.2.3 Block locking

To ensure the correctness of the disk server, we use the `sem_wait` and `sem_post` functions to protect the disk structure. The `mutex` field of the disk structure is used as a semaphore to protect the disk structure.

#### 3.2.4 Persistence and integrity

**Magic headers**: To ensure the integrity of the disk file, we use magic headers to identify the disk file. The magic headers are stored in the first 4 bytes of the disk file. Different kinds of blocks have different magic headers.
**Versioning**: To ensure the persistence of the disk file, we use versioning to avoid confusion when the disk file is updated.

## 4 File System

### 4.1 RemoteDisk

#### 4.1.1 Descriptions

The remote disk module provides as a seamless interface to the basic disk server. When calling its member functions, the remote disk module will send the request to the basic disk server and wait for the response. Also, it would check whether the request has a invalid cylinder or sector number.

### 4.2 Block Manager

#### 4.2.1 Descriptions

The disk does not know the structure of blocks on disk. Hence we need a block manager to manage the allocation and deallocation of disk blocks. The block manager provides the interface for the upper layer to allocate, free, load, unreference and flush blocks. The block manager also acts as a memory pool for the upper layer to cache blocks. The block manager provides the following interface:

```
template <class block_t>
block_t* load(blockid_t block); // Load a block from disk
template <class block_t>
block_t* allocate(blockid_t& block); // Allocate a block
void dirtify(blockid_t block); // Mark a block as dirty
void unref_block(blockid_t block); // Unreference a block
void free_block(blockid_t block); // Free a block
void flush(); // Flush all dirty blocks to disk
```

In our project, we do not manually partition the disk to a inode block section and a data block section. Instead, we use the block manager to manage all disk blocks, which treats all blocks as data blocks.

We define `uint64_t` as the block id type, which is simply composed of the cylinder number as upper 32 bits and the sector number as lower 32 bits.

### 4.2.2 Superblock

The superblock holds the information to manage the disk blocks. The superblock is stored in the first block of the disk. The superblock structure is shown below:

```
struct SuperBlock {
    static constexpr uint32_t MAGIC = 0x2C1D7C0D;
    uint32_t magic; // A magic number to identify the superblock.
    uint32_t block_size; // The size of a block.
    blockid_t free_list_head; // The head of the free list.
    blockid_t root_inode; // The root inode of the file system.
    blockid_t block_end; // The last block used by the file system.
    uint64_t version; // The unix time when the superblock is created.
};
```

The superblock will always have the block id 0, and the root inode will always have the block id 1. However, block id 0 is exposed as an invalid block id to the upper layer, because the block allocation details are invisible to upper layers.

### 4.2.3 Allocation and deallocation

The block manager uses a free list to manage the free blocks. The free list is a singly linked list, and the head of the free list is stored in the superblock. Hence, we need a `FreeBlock` structure to store the free block information:

```
struct FreeBlock {
    static constexpr uint32_t MAGIC = 0x2C1D7C0E;
    uint32_t magic;
    blockid_t next; // The next free block.
    blockid_t id; // The block id of this free block.
    uint64_t version; // The version of the file system who frees this block.
};
```

**Allocation**: When allocating a block, we first check whether the free list is empty. If the free list is empty, we increase the end block and return it. If the free list is not empty, we pop the head of the free list and return it. If the end block has reached the end of the disk, we return an invalid block id since the disk has been full.

**Deallocation**: When deallocating a block, we format the block as a `FreeBlock` structure and push it to the head of the free list.

One problem with free list implementation is that, when freeing blocks, we need to write the freed blocks to disk as they are a part of the free list. This would introduce a performance overhead. To solve this problem, we use caching and lazy flushing mechanism to reduce the number of write operations to disk.

#### 4.2.4 Caching

The block manager provides a memory pool to cache blocks. The block manager uses a hashmap to cache blocks, as well as a free data block list to manage the free data blocks.

**Flush**: We can mamually ask the block manager to flush all dirty blocks to disk.

**Load**: When loading a block, the block manager first checks whether the block is in the cache. If the block is in the cache, the block manager returns the block from the cache. If the block is not in the cache, the block manager loads the block from disk and puts it into the cache.

**Unreference**: When unreferencing a block, the block manager decreases the reference count of the block. If the reference count of the block is zero, the block manager puts the block into the free data block list.

**Lazy flushing**: Actually, the caching mechanism is more complex than the above description. The system caps the memory usage of the block manager with a soft limit and does lazy flushing to disk. That is, when the reference count of a block is zero, we do not immediately flushes the block to disk and deletes it. Instead, we evict not-referenced blocks from the cache to free memory pool when the size of cached data blocks exceeds the soft limit. More advanced algorithms like LRU or LFU can be used to evict blocks, but we use a simple FIFO queue to evict blocks.

**Memory pooling**: The block manager uses a memory pool to cache blocks. The memory pool is implemented as a queue. When a block is evicted, it is put into the tail of the queue. When we need data blocks, we pop the head of the queue. Also, there exists a hard limit for the memory pool, when the size of the memory pool exceeds the hard limit, we will free any blocks to be evicted.

The benefits of these two mechanisms are that we can (1) reduce the number of read and write operations to disk (for example, when we access the same file for multiple times), and (2) reduce the number of newing and deleting data for blocks.

The life cycle of a block is shown in figure 2.

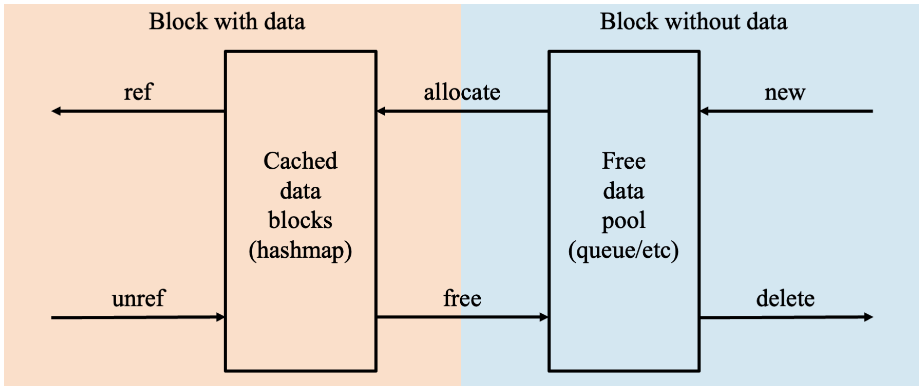

Figure 2: Block Life Cycle

### 4.3 Inode File

#### 4.3.1 Descriptions

The inode file module implements the operations on a single inode file. It provides the interface for the upper layer to create, delete, insert, read, write, and truncate files.

Everything in the file system is a file, including directories. Hence, the inode file module is the basic and middle building block of the file system. The inode file module provides the following interface:

```
bool open(blockid_t inode_id);
blockid_t create(uint32_t owner, uint16_t mode, uint16_t type);
void close();
size_t size() const;
size_t read(char* buf, size_t size, size_t offset);
size_t write(const char* buf, size_t size, size_t offset);
size_t insert(const char* buf, size_t size, size_t offset);
size_t remove(size_t size, size_t offset);
size_t readall(char* buf);
bool removeall();
bool truncate(size_t size);
bool set_mode(uint16_t mode);
bool set_owner(uint32_t owner);
```

### 4.3.2 Inode structure on disk

The inode file is composed of an inode block, a series of entry block (or indirect block) and data blocks. The inode block stores the metadata of the file, such as the owner, mode, type, size, and pointers to the entry blocks and data blocks. The entry block stores pointers to lower level of entries or data blocks, and the data block stores the file data. The structure is shown in figure 3.

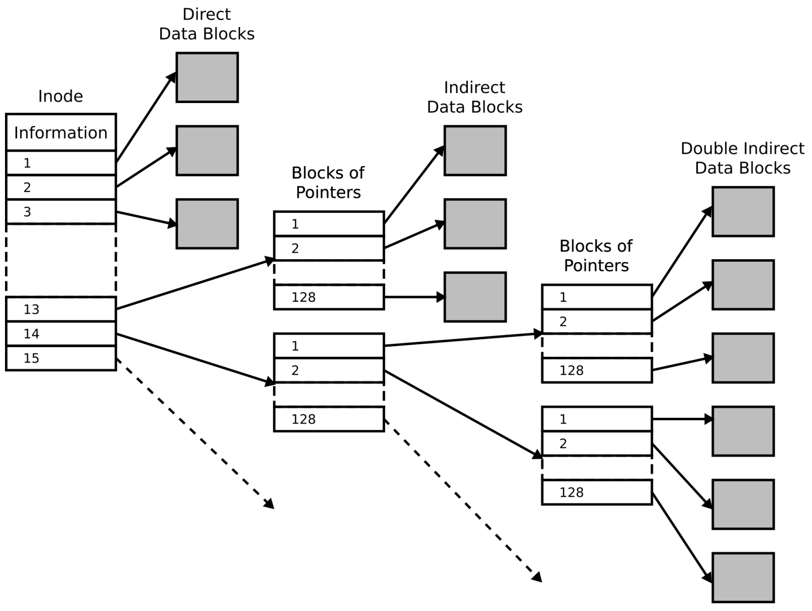

Figure 3: Inode Block Structure

To be specific, the inode block contains 23 direct pointers, 1 indirect, 1 double-indirect, and 1 triple-indirect pointers, to make up 256 bytes. The entry block contains 30 pointers to lower level of entries or data blocks. The data block contains 252 bytes of data, which is the block size subtracted by the size of magic number.

### 4.3.3 Lazy entry updating

When an inode file is active in memory, the inode file class will maintain a data block list. The list is loaded when the file is opened, and all entry blocks are read. All the entries are not updated as we modify the contents of the file. Finally, when the file is closed, the class fill the direct block pointers and entry blocks with points in data block list. To be specific:

**Open**: When opening an inode file, the inode file module loads the inode block from disk. Then, the inode file module loads and caches all entry blocks and reads the data block list.

**Close**: When closing an inode file, the file will fill direct block pointers and entry blocks with points in data block list. When previous allocated entry blocks are not enough, the file will allocate new entry blocks. When there is more entry blocks than needed, the file will free the extra entry blocks.

The purpose of this design is to accelerate the read and write operations at random positions, while circumventing the complexity of maintaining the indirect block trees.

### 4.3.4 File operations

The implementation details of file operations are listed below:

**Create**: When creating an inode file, the inode file module allocates an inode block by the block manager and initializes the inode block.

**Read/Readall**: When reading a file, the file will first check whether the read operation is out of range. Then, the file will dynamically load data block from block manager and read the data from the data block one by one.

**Write**: When writing a file, the file will first check whether the write operation is out of range. Then, the file will dynamically load data block from block manager and write the data.

**Insert/Remove**: Inserting and removing a part of the file is different from the above two operations as they will modify all data blocks after the operation position. To support these two operations, the file will create a temporary data block list, which only supports write operation to accommodate new data. The detailed process for insert is shown in figure 4, and delete is similar to that.

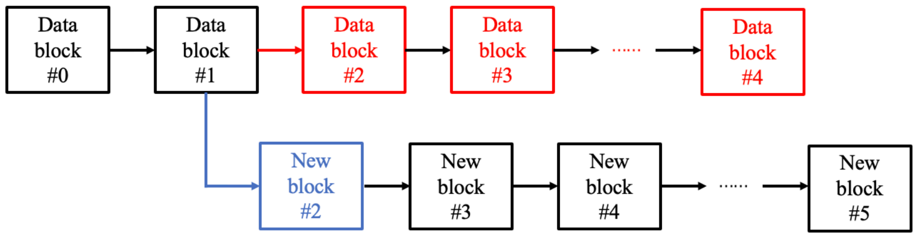

Figure 4: Insert Operation

**Step 1** Create a temporary data block list.

**Step 2** Write new data to the temporary data block list.

**Step 3** Write the old data after the operation position to the temporary data block list.

**Step 4** Replace the old data block list with the temporary data block list.

**Removeall**: Free all data blocks.

**Truncate**: Modify metadata and expand or shrink data block list.

**Set mode/owner**: Modify metadata.

## 4.4 Special files formats

### 4.4.1 Descriptions

The file system maintains two special file formats: the directory and the user information. The directory is a special file format, containing a list of file entries, i.e. file name and inode pointer. The user information is stored in a special file named `userfile` under the root directory. The user information includes user id and user name.

### 4.4.2 Directory

The directories are stored as a special file format, which is a list of file entries. The file entry structure is shown below:

```
struct DirectoryEntry {
    size_t len; // 0 for deleted
    char filename[MAX_FILENAME_LEN];
    blockid_t inode;
};
```

When creating a directory, the file system will create a new inode file and initialize it as a directory. This is done by initializing `Directory` class with a pointer to inode file in memoery, then the file system modifies the directory in memory. When a `Directory` instance destructs it will write the directory entries to the inode file. The `Directory` class provides the following interface:

```
blockid_t lookup(const char* filename) const;
int add_entry(const char* filename, blockid_t inode);
int remove_entry(const char* filename);
int list(std::vector<std::string>& list) const;
```

Directories build up the file system tree structure. A directory saves `.` and `..` entries to represent the current directory and the parent directory. Hence the client can walk through the file system tree by following children and parent pointers.

The root directory and home directory are two special directories in the file system. They are created when the file system is formatted.

### 4.4.3 User information file

The user information is stored in a special file named `userfile` under the root directory. The user information includes user id and user name. The user file is a list of fixed length strings as user names, with their indexes as user ids. The user file is stored in the root directory, and the project implements a in memory cache for the user file. The user file is loaded when the file system is initialized, and the user file is written back to disk when the file system is closed.

The user file provides the following interface:

```
uint32_t add_user(const char* username);
int remove_user(uint32_t uid);
uint32_t lookup(const char* username) const;
const char* get_username(uint32_t uid) const;
int set_username(uint32_t uid, const char* username);
void list_users(std::vector<std::string>& list) const;
```

## 4.5 Support for Multi-users and Multi-clients

### 4.5.1 Descriptions

The file system supports multi-users and multi-clients. The file system maintains user information and permissions, as well as manages locks for concurrency control. The file system provides a context, called `WorkingDir`, for each connecting client, which contains the current directory and the current user. The `WorkingDir` class provides the following interface:

- In order to support concurrency control, the file system ensures only one instance of a file can exist at the same time, that is, two clients accessing the same file are accessing the same memory blocks, along with locks to protect the file. This is done by a internal data structure of file system `FileSystem::node_t`, which contains a pointer to the inode file, a read-write lock and a reference count.
- Provide a context, called `WorkingDir`, for each connecting client, which contains the current directory and the current user. Also, the context checks the permission of the user when accessing files.

### 4.5.2 Internal node

The file system maintains a internal data structure `FileSystem::node_t` to ensure only one instance of a file can exist at the same time. The `node_t` structure is shown below:

```
struct node_t {
    int rwcnt; // pos for read, neg for write
    int refcnt; // number of clients accessing the file
    sem_t sem;
    InodeFile* file;
    Directory* dir; // not null if the file is a directory
};
```

The file system stores the nodes in a hashmap, with the inode id as the key. Hence, the file system ensures that only one instance of a file can exist at the same time. The read-write lock is used to protect the file. The reference count is used to ensure that the file is not deleted when it is being accessed by other clients. The mechanism will be talked about in the following sections.

### 4.5.3 Working directory

The working directory is a context for each connecting client. The working directory contains the current directory and the current user. The working directory provides the following interface:

```
ecode_t create_file(const char* filename);
ecode_t create_dir(const char* dirname);
ecode_t remove(const char* name);
ecode_t remove_dir(const char* dirname);
ecode_t change_dir(const char* path);
ecode_t list_dir(std::vector<std::string>& list);
ecode_t rename(const char* oldname, const char* newname);

ecode_t chmod(const char* name, uint16_t mode);
ecode_t chown(const char* name, uint32_t owner);
ecode_t acquire_file(const char* filename, bool write);
void release_file();
InodeFile& active_file() { return active_file_; }

uint32_t user() const { return user_; }
```

and has the following fields:

```
uint32_t user_;
FileSystem* fs_;
node_t* node_;
InodeFile active_file_;
```

The working directory provides a wrapper to access the current directory and files under concurrency control, which will be discussed in the following sections.

### 4.5.4 Permission control

The project offers per-file and per-user permissions. Every file has a owner and a mode. The mode is a 16-bit integer, with the lower 3 bits representing the read, write, and execute permissions for the owner, the higher 3 bits representing the permissions for other users, which is a simplified version of the unix file permission system. The permission control is implemented in the `test_permission` function. `WorkingDir` class will call this function to test the permission of the user when accessing files.

```
bool test_permission(InodeBlock* inode, uint32_t user, bool write) {
    if (user == 0) return true; // root user
    uint16_t mode; // current access mode
    if (inode->owner != user) {
        mode = write ? FILE_OTHER_WRITE : FILE_OTHER_READ;
    } else {
        mode = write ? FILE_WRITE : FILE_READ;
    }
    return (inode->mode & mode) != 0;
}
```

### 4.5.5 Concurrency control

There are two major concurrency control problems in this project:

- **File lock**: The file system ensures that only one instance of a file can exist at the same time. The file system uses the read-write lock in the `node_t` structure to protect the file. The working directory provides as a wrapper to access current directory and files under the directory. The wrapper would first tries to acquire the lock of the file and directory, then tests the permission of the user, calls the corresponding function of the file system, and finnal releases the lock.
- **Directory lock**: The file system ensures that while a client is accessing a directory, no other client can delete the directory in the recursive `rmdir` operation. In the `rmdir` operation, the file system first tries to acquire the locks of all files and subfolders under the directory, then deletes the files and subfolders, and finally deletes the directory. This mechanism also ensures that a client cannot delete its current directory or its parent directory.

  The same technique is used in the `format` operation, which first tries to acquire the locks of all files and directories under the root directory, thus ensuring that no client can access the file system while the file system is being formatted.

## 5 Experiments

### 5.1 Experiment 1: Basic File and Directory Operations

In this experiment, we test the basic file and directory operations, including creating, deleting, reading, and writing files, as well as creating and deleting directories. We also test the operations of listing files in a directory and changing the current directory.

```
Enter username: root
Login successful
FS >> mkdir test
success
FS >> cd test
success
FS >> mk f
success
FS >> w f 0 testtesttesttesttest
success
FS >> cat f
testtesttesttesttest
FS >> i f 4 -1234567890-
success
FS >> cat f
test-1234567890-testtesttesttest
FS >> d f 10 3
success
FS >> cat f
test-1234590-testtesttesttest
FS >> stat f
InodeFile: inode=6, size=29, owner=0, mode=-wr-wr, type=Regular, nlink=1
A Tue May 28 23:41:29 2024 CST
M Tue May 28 23:41:21 2024 CST
C Tue May 28 23:40:07 2024 CST
Datablocks: id= 7
FS >> mkdir dir
success
FS >> cd ..
success
FS >> ls
total: 5
.
..
userfile
home
test
FS >> cd test
success
FS >> ls
```

```
total: 4
.
..
f
dir
FS >> rmdir dir
success
FS >> rm f
success
FS >> cd ..
success
FS >> rmdir test
success
FS >> ls
total: 4
.
..
userfile
home
```

We can see that the file system supports basic file and directory operations. The file system can create, delete, read, and write files, as well as create and delete directories. The file system can also list files in a directory and change the current directory.

### 5.2 Experiment 2: Multi-user Operations

In this experiment, we test the multi-user operations, including creating users and assigning different permissions options to files and directories. First, we create two new users under root user:

```
Enter username: root
Login successful
FS >> adduser user1
success
FS >> adduser user2
success
FS >> lsuser
total: 2
1:user1
2:user2
FS >> exit
```

Then, we login as user1 to create a file and set the permission of the file to be read-only for user2:

```
Enter username: user1
Login successful
FS >> cd home
success
FS >> mk f
success
FS >> chmod f 11
success
FS >> w f 0 hello!
success
FS >> stat f
InodeFile: inode=4, size=6, owner=1, mode=--r-wr, type=Regular, nlink=1
A Wed May 29 00:51:20 2024 CST
M Wed May 29 00:51:18 2024 CST
C Wed May 29 00:51:06 2024 CST
Datablocks: id= 5
FS >> exit
```

Finally, we login as user2 to try to read and write the file:

```
Enter username: user2
Login successful

FS >> cd home
success
FS >> cat f
hello!
FS >> i f 2 --
Error (-9): Permission denied
FS >>
```

We can see that the file system supports multi-user operations. The file system can create users and assign different permissions options to files and directories.

### 5.3 Experiment 3: Multi-client Concurrent Operations

In this experiment, we test the multi-client concurrent operations. We run two clients concurrently, and one client creates a file and writes to the file, while the other client reads the file. The file system should ensure that only one instance of a file can exist at the same time, and the file system should ensure the correctness of the file system. Also, we will test the `rmdir` and `format` operations. Suppose that user1 and user2 are two clients.

```
Enter username: user1
Login successful
FS >> cd home
success
FS >> ls
total: 3
.
..
f

FS >> w f 0 ------
success
FS >> cat f
------
FS >> rmdir dir
Error (-14): Device or resource busy
```

```
Enter username: user2
Login successful
FS >> cd home
success
FS >> cat f
------
FS >> mkdir dir
success
FS >> cd dir
success
FS >> mk f
success
FS >> i f 0 1234
success
FS >> format
Error (-14): Device or resource busy
```

We can see that the file system supports multi-client concurrent operations. The file system ensures that only one instance of a file can exist at the same time, and the concurrency control the correctness of the file system under multi-client scenes.

## 5.4 Experiment 4: Large Files

In this experiment, we test the file system with large files. We create a file with a size of 64KB and perform read and write operations on the file.

```
Enter username: root
Login successful
FS >> cd home
success
FS >> mk f
success
FS >> trunc f 65536
success
FS >> stat f
InodeFile: inode=4, size=65536, owner=0, mode=-wr-wr, type=Regular, nlink=1
A Wed May 29 01:15:56 2024 CST
M Wed May 29 01:15:53 2024 CST
C Wed May 29 01:15:48 2024 CST
```

```
Datablocks: id= 5 6 7 8 9 <omitted>
EntryBlock: id=20000000a, count=30, parent=0
    Children id= 1c 1d 1e 1f 20 21 22 23 <omitted>
EntryBlock: id=20000000b, count=7, parent=0
    Children id= 20000000c 20000000d 20000000e 20000000f <omitted>
EntryBlock: id=20000000c, count=30, parent=0
    Children id= 3a 3b 3c 3d <omitted>
EntryBlock: id=20000000d, count=30, parent=0
    Children id= 58 59 5a 5b 5c <omitted>
EntryBlock: id=20000000e, count=30, parent=0
    Children id= 76 77 78 79 7a 7b 7c 7d 7e 7f <omitted>
FS >> w f 20494 hidden_data
success
FS >> i f 20502 --insert--
success
FS >> r f 20494 40
hidden_d--insert--ata
```

## 6 Conclusion

In this project, we implemented a simple file system based on the inode structure. The file system supports basic file and directory operations, as well as multi-user operations and multi-client concurrent operations. The file system is implemented in the user space, and the file system is implemented as a user-level file system. The concurrency control mechanism is implemented using the unix semaphores.

From the beginning of designing the file system, we choose to use free list as the strategy to manage blocks. However, in the middle of the implementation we realized that free list introduce tremendous overhead when freeing blocks. To solve this problem, we use caching and lazy flushing mechanism to reduce the number of write operations to disk.

The overall process of designing, finding issues, and solving problems is a valuable experience. We learned a lot from this project, and we believe that the knowledge we learned from this project will be helpful in our future study and work.