Project 3 Report

王智烨 521021911051

Project Files

```
Prj3+521021911051
    step1
        disk.c
        disk.log
        disk_storage_file
    step2
        fs.c
        fs.log
        filesystem
    step3
        client.c
        disk.c
        disk.log
        disk_storage_file
        fs.c
        fs.log
        filesystem
    report.pdf
    report.md
    Typescript.pdf
    Prj3README
```

Experiments

1. Design and implement a basic disk-like secondary storage server.
2. Design and implement a basic file system to act as a client using the disk services provided by the server designed above.
3. Use socket API.

Step1: Design a basic disk-storage system

Introduction

Command line:

```
./disk <#cylinders> <#sector per cylinder> <track-to-track delay> <#disk-storage-filename>
```

The disk system also supports the following commands:

```
I
//Information request. The disk returns two integers representing the disk
geometry: the number of cylinders and the number of sectors per cylinder.
R c s
//Read request for the contents of cylinder c sector s. The disk returns Yes
followed by a writespace and those 256 bytes of information, or No if no such
block exists
W c s
//Write a request for cylinder c sector s. The disk returns Yes and writes the
data to cylinder c sector s if it is a valid write request or returns No
otherwise.
E
//Exit the disk-storage system.
```

This disk system requires two files, `disk_storage_file` and `disk.log`, in addition to the source code during execution. `disk_storage_file` serves as a real disk file to store all the data, and the `disk.log` is used to record the the results of each step of the program during operation.

The output format is listed below:

```
I
//output two integers, separate by a writespace, which denotes the number of
cylinders and the number of sectors per cylinder.
R c s
//output No if no such block exists. Otherwise, output Yes followed by a
writespace and those 256 bytes of information.
W c s data
//Write Yes or No to show whether it is a valid write request or not.
E
//output Goodbye!.
```

Analysis

- `mmap()`

The core of the step1 is that we should master the use of `mmap()`.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

`mmap()` is a system call in Unix and Unix-like systems that establishes a mapping between the process address space and a file. It allows for memory-mapping files, enabling direct access to the file's contents as if accessing memory, without the need for traditional read and write operations.

Simply put, when we use this function, we can close the file and read and write the contents of the file via `memcpy()`.

- Sectors、blocks and Cylinders

Another important concept is how to arrange `disk_storage_file` to make it look like a real disk. The user uses the `c` and `s` parameters to simulate writing data on the disk. Since we are actually writing the program to `disk_storage_file`, we need to compute the actual address of the data being written.

$$disk file[BLOCKSIZE * (c * SECTORS + s)]$$

- `lseek()`

```
off_t lseek(int fd, off_t offset, int whence);
```

The `lseek` function is a system call in Unix and Unix-like operating systems that is used to reposition the file offset of an open file. It allows you to move the read/write position within a file, enabling random access to different parts of the file.

However, we should note that if we use · `lseek()` the first tome to stretch the file, we should also use `write()` to check that the file is stretched successfully.

In my initial attempts, I didn't use `write()`, which led to errors in my subsequent use of `disk_storage_file`.

Step2: Design a basic file system

Introduction

Implement a file system. The file system should provide operations such as: initialize the file system, create a file, read the data from the file, write a file with given data, append data to a file, change data in a file, remove a file, create directories, etc.

Command line:

```
./fs
```

To simulate a real filesystem, our program also needs to support the following instructions:

```
f
//Format. This will format the file system on the disk, by initializing any/all
of the tables that the file system relies on
```

```
mk f
//Create file. This will create a file named f in the file system.
```

```
mkdir d
// Create directory. This will create a subdirectory named d in the current
directory.
```

```
rm f
// Delete file. This will delete the file named f from the current directory
```

```
cd path
// Change directory. This will change the current working directory to the path.
The path is in the format in Linux, which can be a relative or absolute path.
When the file system starts, the initial working path is /. You need to handle .
and .. .
```

```
rmdir d
// Delete directory. This will delete the directory named d in the current
directory.
```

```
ls
// Directory listing. This will return a listing of the files and directories in
the current directory.You are also required to output some other information,
such as file size, last update time, etc.
```

```
cat f
//Catch file. This will read the file named f, and return the data that came from
it.
```

```
w f l data
// Write file. This will overwrite the contents of the file named f with the l
bytes of data. If the new data is longer than the data previously in the file,
the file will be made longer. If the new data is shorter than the data previously
in the file, the file will be truncated to the new length.
```

```
i f pos data
// Insert to a file. This will insert into the file at the position before the
posth character(0-indexed), with the l bytes of data. If the pos is larger than
the size of the file. Just insert it at the end of the file.
```

```
d f pos l
// Delete in the file. This will delete contents from the pos character (0-
indexed), delete l bytes, or till the end of the file
```

Like the disk system, the file system has its own log file, which writes "Done" for the `f` command, "Goodbye" for the `e` command, Yes for the `ls` command after successful execution, and the size and time information of the files and subdirectories in order of creation time.

Analysis

Structure Arrangement

It is very important for a file system to arrange and allocate the storage units in the file system. I used Inode instead of FAT architecture, and I will describe my file system structure layout below.

First, I classify the basic unit of storage as block, whose size is 256 bytes, which corresponds to the disk system and facilitates the subsequent combination.

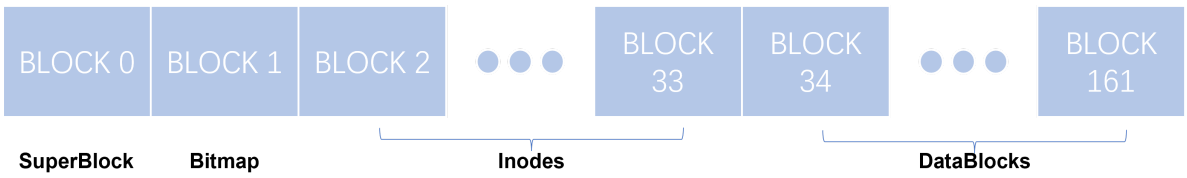

Figure 1 File system structure

- SuperBlock

Superblock stores important information about the file system. Here I chose to record the virtual address of the root directory of the filesystem, the total number and available number of inodes and datablocks.

```
typedef struct
{
    _u32 s_inodes_count;
    _u32 s_blocks_count;
    _u32 s_free_inodes_count;
    _u32 s_free_blocks_count;
    _u32 s_root;
}SUPERBLOCK;
```

- Bitmap

  Bitmap is a structure that records the state of all inodes and datablocks by bit. Available storage locations are represented by a binary 0, and those already used are represented by a binary 1.

```
typedef struct //64 bits for INODE 128 bits for BLOCKS Totaoly 24 bytes
{
    _u32 inodeMap[2];
    _u32 blockMap[4];
}BITMAP;
```

  Since we have 64 inodes and 128 datablocks, I implement a 32-bit unsigned integer array of 2 elements and a 32-bit unsigned integer array of 4 elements to record the state of each inode and datablock bit by bit.

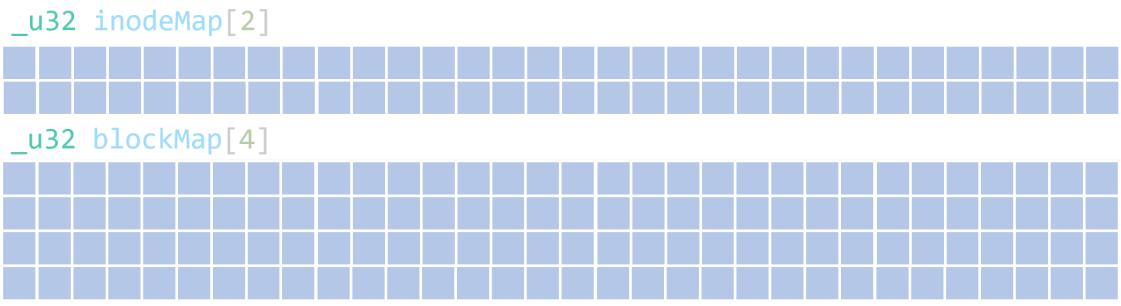

Figure 2 Bitmap

- Inode

  Inodes are used to manage files and directories in the file system. Each file and directory has a unique inode in the file system.

The inode records metadata about the file or directory, including information about the file's permissions, owner, size, timestamp, and the location of the data block on the disk. It is similar to the "fingerprint" of a file, and the inode can be used by the operating file system to find and access the actual contents of the file.

```
typedef struct  //Inode 64 bytes
{
    _u32 i_id;//Inode ID
    _u32 i_mode;//directorty file or file 1 for directory 0 for file
    _u32 i_uid;//num of file
    _u32 i_gid;//father directory, record the inode
    _u32 i_size;
    _u32 i_access;
    _u32 i_blocknums;
    _u32 i_ctime;//create time;
    _u32 i_mtime;//modified time;
    _u32 i_atime;//access time;
    _u32 i_direct[4];
    _u32 i_sindirect;
    _u32 i_dindirect;//son directory 32  is max number
}INODE;
```

I set the size of the Inode to 64btytes, which guarantees that 4 Inodes can be stored in a block.

In linux systems, since everything is a file, files that are essentially directories also use this Inode structure. I use `i_mode` to identify whether this Inode is a normal file or a directory file.

Normal file:

For normal files, `i_mode` is set to 0, `i_uid`, `i_gid` are used to record the userID and groupID of the file (this feature is not implemented due to time), `i_size` records the size of the file, `i_access` records the access rights of the file, and `i_blocknums` records the number of blocks used. The datablock is stored in `i_direct`, `i_indirect` and `i_dindirect`.

Directory file :

For directory files, since they use the same inode as normal files, the meanings represented by some member variables will change.

First, `i_mode` will be set to 1 to indicate that this is a directory file, `i_uid` will be used to indicate the number of files in the current directory, `i_gid` will be used to record the inode number of the parent directory, and `i_dindirect` will be used to record the inode number of the subdirectory.
Among the inodes, the inode number of each file is unique and it is recorded in the `i_id`.
The name of each common file is recorded in the datablock of the directory file, where the name of the current directory is recorded first.

- FileDirectory

  Filedirectory is a struct contained filename and its inode number, helping organize the information stored in the directory file datablock.

```
typedef struct //28 bytes in all. So a block can hold 8 files inall.
{
    _u32 f_inodeNum;
    char f_fileName[24];
}FILEDIRECTORY;
```

Just as what have been introduced in *Directory file*, the datablock of directory file is used to store the name and id of files in it, which is arranged in the form of struct `FILEDIRECTORY`. Also, the `FILEDIRECTORY` of the directory is always stored at the first position.

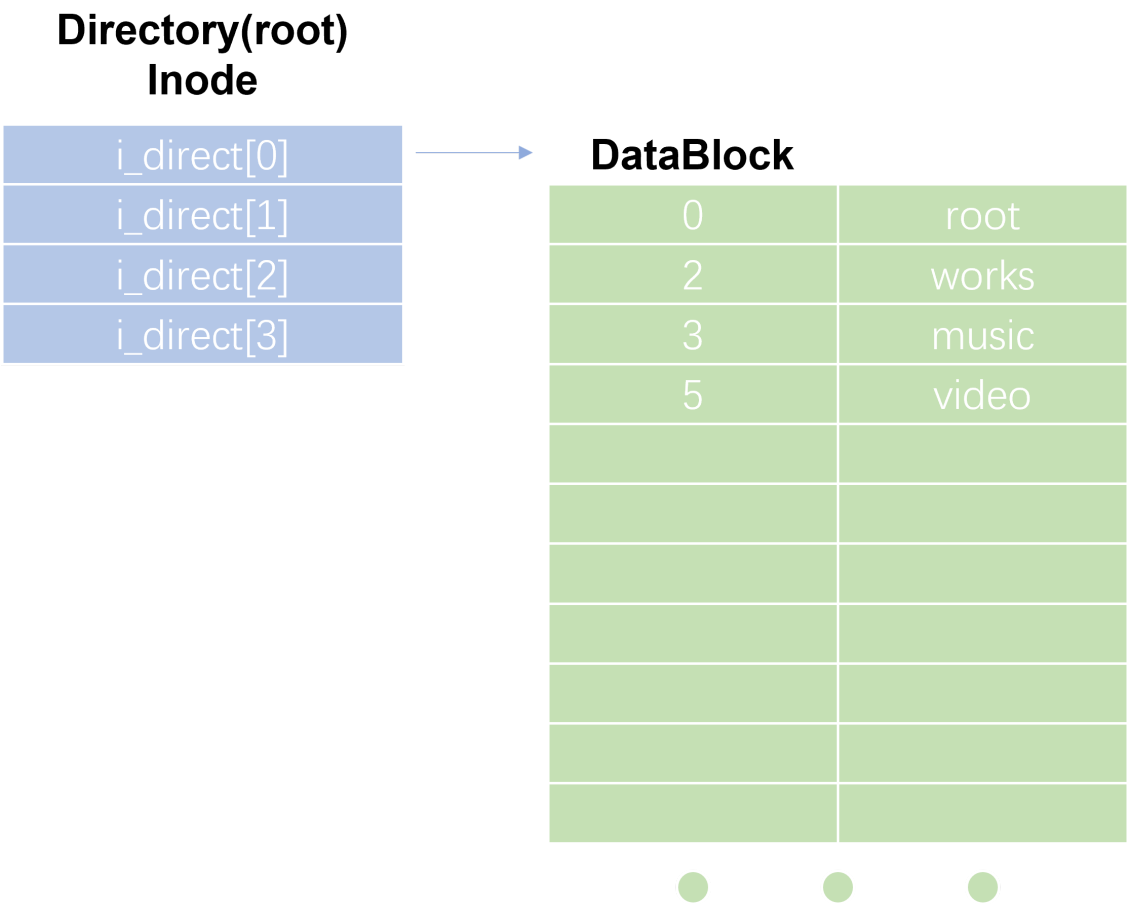

Figure 3 FileDirectory

- SingleIndirect

  This struct is designed to serve as single indirect table. For a single indirect table, the datablock number is stored in this table. We set the table can record the information of 8 blocks(you may wonder that why this number is so small, this is because if I maximaize the utilization of this table, there may not be enough datablocks, we just have 128 datablocks in all).

```
typedef struct //256 bytes ; just want it to point to 8 blocks
{
    _u32 s_blocknum[8];
    _u32 available[8];
}SINGLEINDIRECT;
```

What should be mentioned is that although I design the array `available[8]` to help record the status of each `blocknum`, it didn't work when I actually write the program. Because I will initialize this struct to set each `s_blocknum` to -1, I can check the status to see whether it equals to -1.

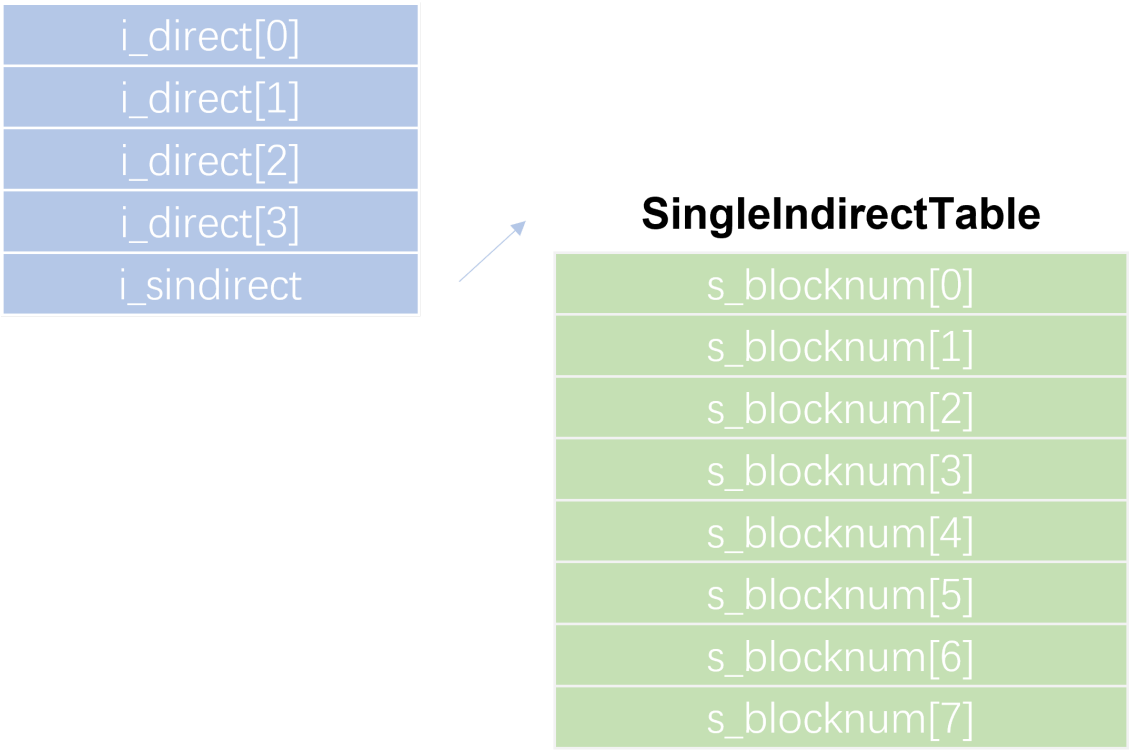

Figure 4 SingleIndirect Table

- SondirectoryTable

SondirectoryTable is designed to record subdirectories in directory file, which is stored in `i_dindirect`.

```
typedef struct //136 - 256 bytes, stored in i_dindirect in directory file, for
store son directory.
{
    _u32 sonDirectory[32];// 128 bytes
    _u32 map;
    _u32 num;
}SONDIRECTOPRYTABLE;
```

A directory can hold up to 32 subdirectories, the inode number of which stores in array `sonDirectoy`. Like bitmap, a member variable is called map, which use bit to record the status of each `sonDirectory`. The design will accelerate the process of deleting or adding subdirectory. The member variable `num` is used to record the amount of subdirectory.

Address

Perhaps you have noticed that in my program, `inodeId` and `blockId` are important representations of inode and datablock locations, respectively. In fact, they are both virtual addresses, and how we should translate them to physical addresses on disk is something that must be considered.

In order to solve this problem, I set two const variables representing the offset address of inode and datablock. With the help of them, we can easily transfer the virtual address to physical address.

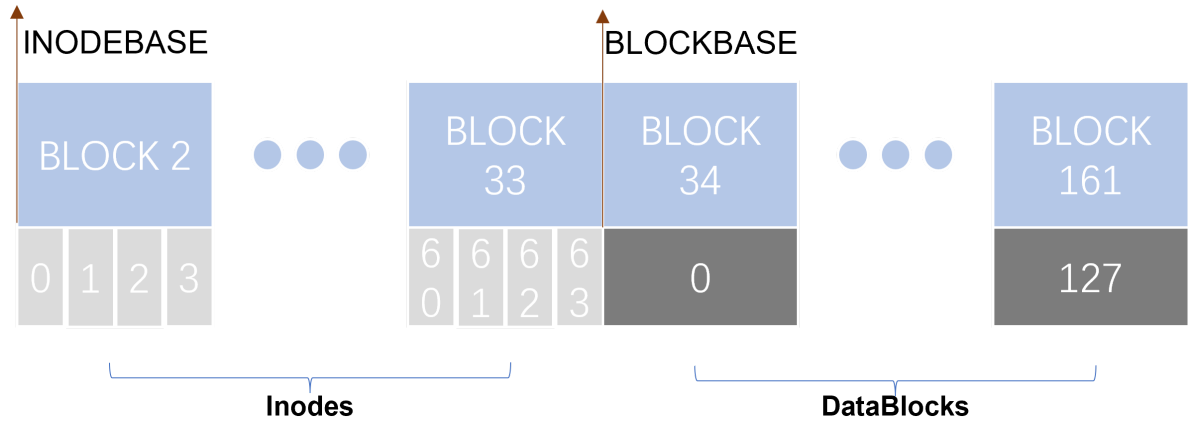

Figure 5 INODEBASE and BLOCKBASE

For example, if the `inodeID` equals to `i`, we copy this inode to disk in this way:

$$memcpy(\&diskfile[INODEBASE * BLOCKSIZE + i * INODESIZE], buf, INODESIZE);$$

If the `blockID` equals to `b`, we copy this datablock to disk in this way:

$$memcpy(\&diskfile[(BLOCKBASE + b) * BLOCKSIZE], buf, BLOCKSZIE);$$

Read and Write

No matter storing the struct or the data into the disk or reading from the disk, we should handle the problem that how to in which form these information is stored.

I choose to transfer everything into `char*`. In this way, I can solve the problem by using the `memcpy()` function for both reading and writing.

```
struct mystruct
//write to disk
char* buf = (char*)&mystruct;
memcpy(&diskfile[INODEBASE * BLOCKSIZE + i * INODESIZE], buf, INODESIZE);
//read fom disk
memcpy((char*)&mystruct, &diskfile[INODEBASE * BLOCKSIZE + i * INODESIZE],
INODESIZE);
```

Delete file and directory

The logic I delete file and directory is different. Because I want the files to be stored in the order they were created, but not the directories.

For the files, when I delete one of the files, all the files after that file will move forward to fill the empty space of that file.

For the directories, when I delete the directory, I only change the corresponding `blocknum` in `stb` to -1 and make the corresponding operation in `bitmap` and `map` in `stb`.

Comparing these two methods, the deletion of files consumes more time, but when adding new files it is only necessary to add them at the end. The deletion of directories is faster and only needs to find a location with a binary value of 0 on the map when adding. However, this method will not keep the order of the contents inside.

Current directory

To know which directory the user is in is important to a file system, I set two global variables to solve this problem.

```
INODE currentDirectory;
_u32 currentDirectoryInode;
```

To avoid frequent visit to disk to modify or get the current directory, I store the latest information in `currentDirectory` and the id in `currentDirectoryInode`. Only when the user need to switch current directory will I store the `currentDirectory` back and read new `currentDirectory`.

Function

Here I would list all the functions I design.

```
//delete 10 13 in string
void delchar(char* str, const size_t max_len);
//initialize sondirectory table
void init_SONDIRECTORYTABLE(int blockPos);
//initialize superblock
void init_SUPERBLOCK();
//initialize single indirect table
void init_SINGLEINDIRECT(int blockPos);
//set a bit bitmap to 1
void setBitMap(int type, int N);
//initialize bitmap
void init_BITMAP();
//initialize an inode
void init_INODE(INODE *i);
//update the time of an inode, c for create, a for access, m for modify
void timeUpdate(INODE *i, int c, int a, int m);
//check some bit in bitmap
int searchBitMap(int type);
//set the bit in bitmap to 0 again, free the space
void returnBitMap(int type, int N);
//add a filedirectory to current directory
int addFiletoDirectory(int fileInode, char* fileName);
//initialize the filesystem
void init_FS(char* diskfile);
```

```
//create a file: mk f
int createFile(char* fileName);
//link a subdirectory to its father directory
int fatherLink (int sonInode);
//create a directory: mkdir f
int createDirectory(char* directoryName);
//delete a file in current directory
int deleteFile(char * deleteFile);
//search the sondirectory ID of current directory
int searchInDir(char* name);
//delete the sondirectory of current directory
int deleteDir(char* name);
//list, refer to the datablock of current directory: ls
int listing(FILE* fp);
//search a file ID in current directory
int searchFile(char* name);
//clear a file, set all the datablock to 0
void clearFile(int inodeNum);
//return the min
int min(int a, int b);
//write the file from some pos, to some dest
int writeFile(int inodeNum, int len, char* data, int beginPos);
//write file: w
int writeIn(char* name, int len, char* data);
//insert file: i
int insertIn(char* name, int pos, int len, char* data);
//delete the content: d
int deleteIn(char* name, int pos, int len);
```

Step 3: Work together

Introduction

Command line:

```
./disk <#cylinders> <#sectors per cylinder> <track-to-track delay> <#disk-storage-filename> <DiskPort>
./fs <DiskPort> <FSPort>
./client <FSPort>
```

As in step1 and step2, disk.log and fs.log record the status of each step of the program. In addition, for debugging and ease of use, I also print a lot of information from the command line.

Besides the requirement in the *Project3description*, I add a new command `E` to Shut down disks, file systems and clients in sequence.

Analysis

Client and file system communication

In the communication between the client and the file system, I specify a protocol in which the client first sends a message to the file system in one round, followed by the file system sending a message to the client.

The message that the client sends to the file system is actually the command we typed on the command line in step2, so we don't need to do much processing here. The information sent by the file system is generally the result of execution of the command, except for the E command, where the file system will send back the specified content for the client.

```
while(1)
    {
        printf("Please type in the command\n");
        fgets(buffer, 2048, stdin);
        n = write (sockfd, buffer, strlen(buffer));
        if ( n < 0)
            herror("Error: fail to wirte to socket\n");
        bzero(buffer, 2048);
        n = read(sockfd, buffer, 2047);
        if(n < 0)
            herror("Error: fail to read from the socket\n");
        if (strcmp(buffer, "E\n") == 0)
        {
            printf("Goodbye!\n");
            close(sockfd);
            return 0;
        }
        printf("%s", buffer);
        bzero(buffer, 2048);
    }
```

File system and disk communication

In step2, I store the contents of the file system in the disk in the file system, in step3, it is equivalent to me using the disk in the disk system. this means I need to change all the `memcpy()`, `memset()` functions in the file system and specify the relevant communication protocols.

```
typedef struct
{
    int rw;//0 for r, 1 for w ;2 for formatted; 3 for move; 4 for clear; 5 for E
    int beginPos;
    int size;
    int oriPos;
}TRANS;
```

First, I'd like to introduce the struct `TRANS`, which is an important part of the communication protocols. `rw` indicates the operation to the disk: 0 for reading from disk, 1 for writing into disk, 2 for formatting the disk; 3 for moving something in disk; 4 for clearing something in disk ; 5 for exit the disk; `beginPos` is the begin position of the operation. `size` is data size you need to operate in disk. `oriPos` is only for move operation, recording the origin position of the data.

- writeDisk

```
//fs.c
//First send TRANS, and check 1\n; then send data, then check 1\n
int writeDisk(int diskPos, char* data, int size)
{
    //write
```

```
    TRANS t;
    init_TRANS(&t);
    t.rw = 1;
    t.beginPos = diskPos;
    t.size = size;
    size_t len = strlen(data); // 获取 buf 的长度
    char buf[sizeof(TRANS)];
    memcpy(buf, &t, sizeof(TRANS));
    int n = write(sockfd2, buf, sizeof(TRANS));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    bzero(buf, sizeof(TRANS));
    n = read(sockfd2, buf, sizeof(buf));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    if (strcmp(buf, "1\n") == 0)
    {
        n = write (sockfd2, data, t.size);
        if(n < 0)
        {
            printf("Error: fail to communicate with disk\n");
            return 1;
        }
        bzero(buf, sizeof(TRANS));
        n = read(sockfd2, buf, sizeof(buf));
        if (strcmp(buf, "1\n") != 0)
        {
            printf("Error: fail to write in\n");
            return 1;
        }
        return 0;
    }
    else
    {
        printf("Error: fail to write in\n");
        return 1;
    }
    return 0;
}
```

```
//disk.c
case 1://write
if (formatted != 1 )
{
    n = write(newsockfd, "0\n", strlen("0\n"));
    printf("Error: not formatted yet\n");
    break;
}
n = write(newsockfd, "1\n", strlen("1\n"));
```

```
if (n < 0)
{
    printf("Error: fail to communicate with disk\n");
    break;
}
bzero(buf,sizeof(TRANS));

content = malloc(t.size * sizeof(char));
n = read(newsockfd, content, t.size);
if (n < 0)
{
    printf("Error: fail to communicate with disk\n");
    break;
}
memcpy(&diskfile[t.beginPos], content, t.size);
n = write(newsockfd, "1\n", strlen("1\n"));
fprintf(fp, "Yes\n");
free(content);
break;
```

The communication protocol is portrayed below:

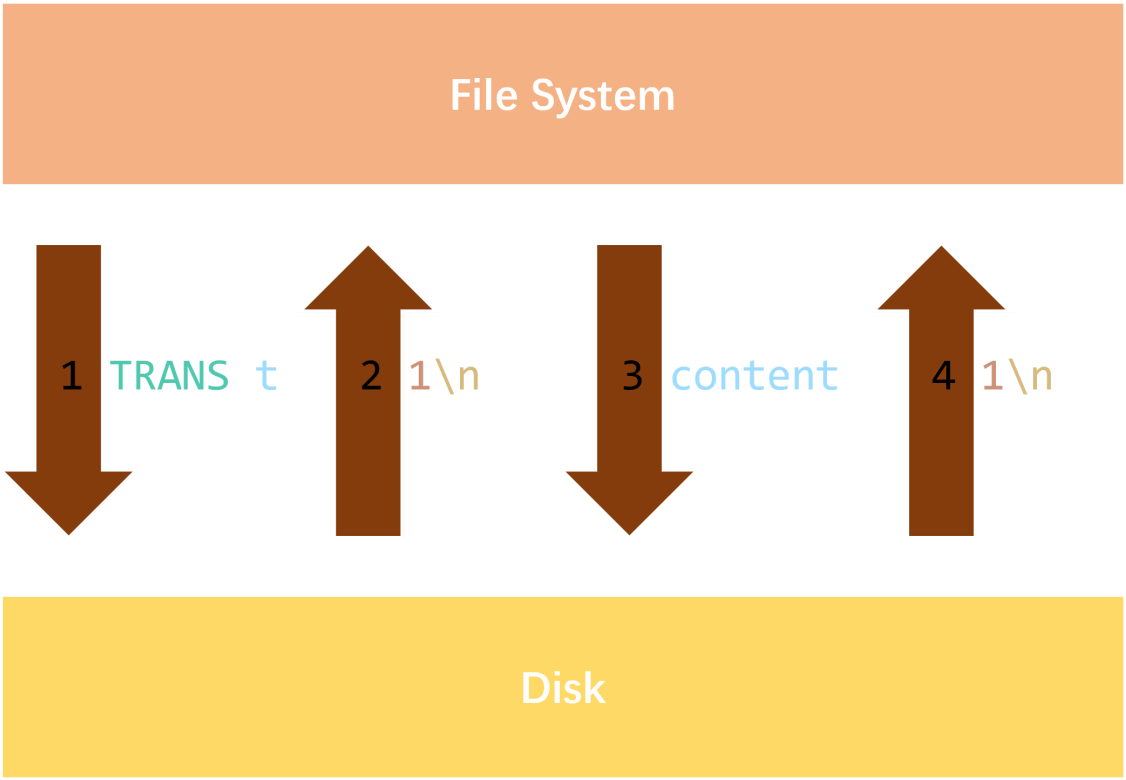

Figure 6 writeDisk communication

- readDisk

```
//fs.c
```

```
int readDisk(char* data, int diskPos, int size)
{
    TRANS t;
    init_TRANS(&t);
    t.rw = 0;
    t.beginPos = diskPos;
    t.size = size;
    char buf[sizeof(TRANS)];
    //printf("RD:%d size :%d\n", t.beginPos,t.size);
    memcpy(buf, &t, sizeof(TRANS));
    int n = write(sockfd2, buf, sizeof(TRANS));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    bzero(buf, sizeof(TRANS));
    n = read(sockfd2, data, t.size);
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }

    return 0;
}
```

```
//disk.c
case 0://read
if (formatted != 1 )
{
    printf("Error: not formatted yet\n");
    break;
}

content = malloc(t.size * sizeof(char));
memcpy(content, &diskfile[t.beginPos], t.size);

n = write(newsockfd, content, t.size);
if (n < 0)
{
    printf("Error: fail to communicate with disk\n");
}
free(content);
fprintf(fp, "Yes\n");
break;
```

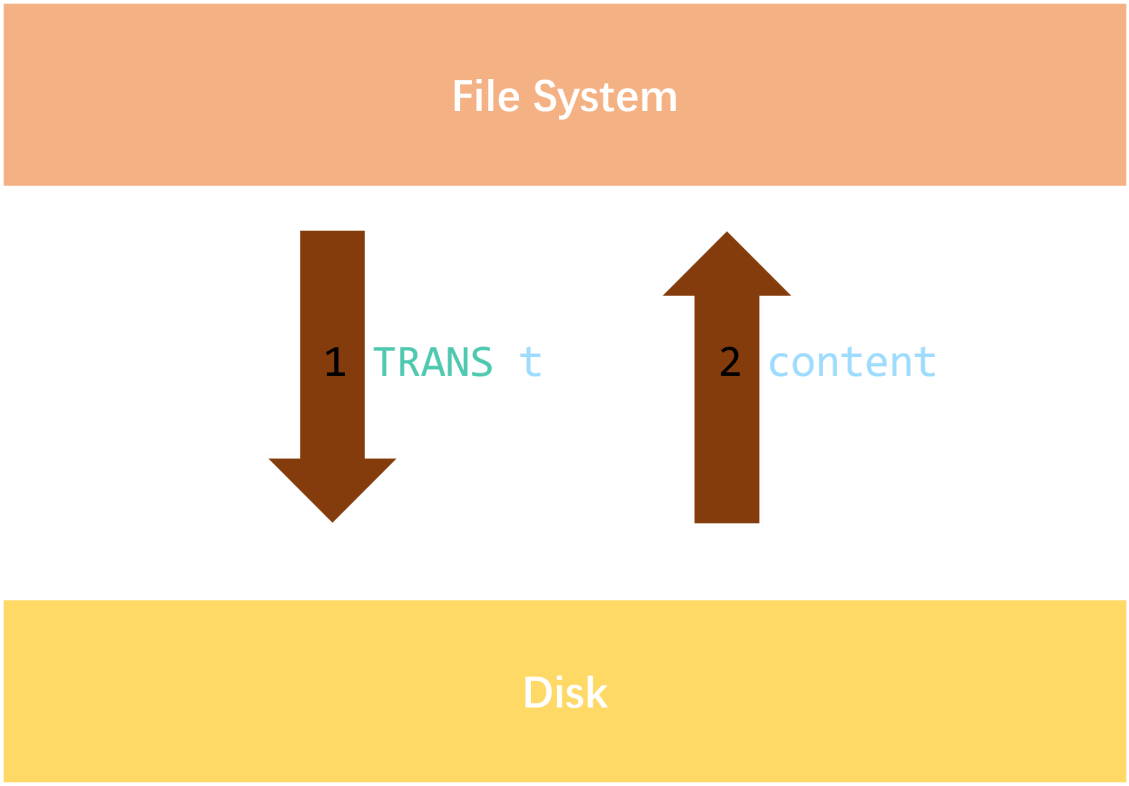

Figure 7 readDisk communication

- moveDisk

```
//fs.c
int moveDisk(int destPos, int oriPos, int size)
{
    TRANS t;
    init_TRANS(&t);
    t.rw = 3;
    t.beginPos = destPos;
    t.oriPos = oriPos;
    t.size = size;
    //printf("MD:%d size :%d\n", t.beginPos,t.size);
    char buf[sizeof(TRANS)];
    memcpy(buf, &t, sizeof(TRANS));
    int n = write(sockfd2, buf, sizeof(TRANS));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    bzero(buf, sizeof(TRANS));
    n = read(sockfd2, buf, sizeof(buf));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    if (strcmp(buf, "1\n") == 0)
        return 0;
```

```
    else
    {
        printf("Error: fail to write in\n");
        return 1;
    }
    return 0;
}
```

```
//disk.c
case 3://move
if (formatted != 1 )
{
    n = write(newsockfd, "0\n", strlen("0\n"));
    printf("Error: not formatted yet\n");
    break;
}
memcpy(&diskfile[t.beginPos], &diskfile[t.oriPos], t.size);
n = write(newsockfd, "1\n", strlen("1\n"));
if (n < 0)
{
    printf("Error: fail to communicate with disk\n");
}
break;
```

- clearDIsk

```
//fs.c
int clearDisk(int beginPos,int a,int size)
{
    TRANS t;
    init_TRANS(&t);
    t.rw = 4;
    t.beginPos = beginPos;
    t.size = size;
    //printf("CD:%d size :%d\n", t.beginPos,t.size);
    char buf[sizeof(TRANS)];
    memcpy(buf, &t, sizeof(TRANS));
    int n = write(sockfd2, buf, sizeof(TRANS));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    bzero(buf, sizeof(TRANS));
    n = read(sockfd2, buf, sizeof(buf));
    if (n < 0)
    {
        printf("Error: fail to communicate with disk\n");
        return 1;
    }
    if (strcmp(buf, "1\n") == 0)
        return 0;
    else
    {
```

```
        printf("Error: fail to write in\n");
        return 1;
    }
    return 0;
}
```

```
//disk.c
case 4://clear
if (formatted != 1 )
{
    n = write(newsockfd, "0\n", strlen("0\n"));
    printf("Error: not formatted yet\n");
    break;
}
memset(&diskfile[t.beginPos], 0, t.size);
n = write(newsockfd, "1\n", strlen("1\n"));
if (n < 0)
{
    printf("Error: fail to communicate with disk\n");
}
break;
```

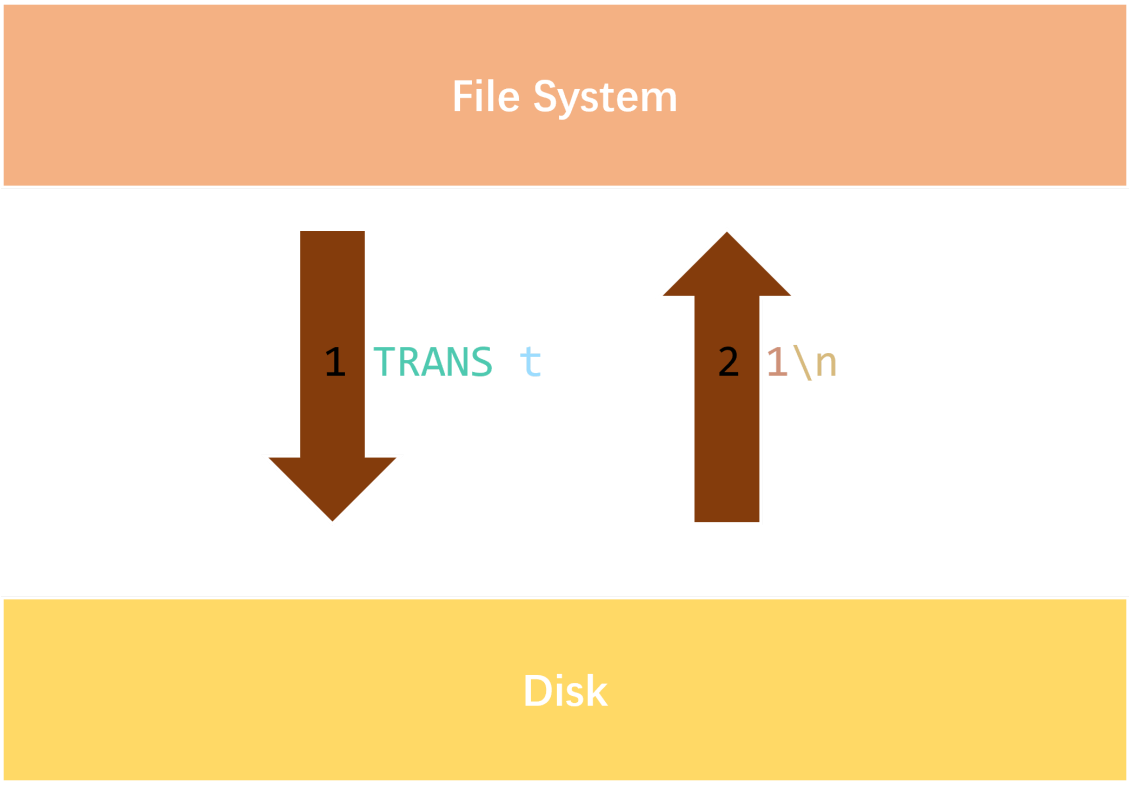

Figure 8 moveDisk&clearDisk communication

Handle the shut down

In the step 3, we need to handle the sudden shut down of file system and the client.

Through inquiry and study I learned that the return number of `write()` and `read()` wil be 0 if the communication is shut down. So I add some related judge to make sure the shut down can be detected.

```
else if(n == 0)
{
    printf("Oops!Connection failed.\n");
    newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);
    printf("New connection is built.\n");
}
```

However, if file system is shut down, the scenario becomes a little more complicated. Just as what I have mentioned, to reduce the frequent visit of the disk, I add global variables `currentDirectory` and `currentDirectoryInode`. So when the file system is connected again, the client is asked to send `r` command which means to recover the file system. In this way, file system will reload the disk and load the `currentDirectory` to the root directory.

```
else if (strcmp(line, "r\n") == 0)
{
    readDisk((char*)&sb, 0, sizeof(SUPERBLOCK));
    readDisk((char*)&bmp, 1*BLOCKSIZE, sizeof(bmp));
    readDisk((char*)&currentDirectory, INODEBASE* BLOCKSIZE, INODESIZE);
    currentDirectoryInode = currentDirectory.i_id;
    printf("currentDirectory.i_id:%u,
currentDirectory.i_uid:%u\n",currentDirectory.i_id, currentDirectory.i_uid);
    printf("Filesystem is recovered to Root.\n");
    fprintf(fp,"Recover\n");
    int n = write (newsockfd1, "Recover\n", strlen("Recover\n"));
    if (n < 0)
        herror("Error: fail to pass\n");

}
```