# Linux简单文件系统设计

## 项目概述

实现一个模拟的文件系统，实现创建，删除，打开读取，读写等操作。

## 项目分析

使用struct结构体创建数据结构构建超级快，文件信息，文件描述符信息。使用dir\_index描述dir目录的信息，使用dir\_len描述目录长度，使用data\_index描述文件数据的位置。实现make\_fs进行文件系统的创建，mount\_fs进行文件系统的挂载，umount\_fs进行文件系统的卸载，fs\_open打开文件，fs\_close关闭文件等方法。

## 项目实施

### 程序源代码

Disk.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include "disk.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int active = 0; /\* is the virtual disk open (active) \*/

static int handle; /\* file handle to virtual disk \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int make\_disk(char \*name)

{

int f, cnt;

char buf[BLOCK\_SIZE];

if (!name) {

fprintf(stderr, "make\_disk: invalid file name\n");

return -1;

}

if ((f = open(name, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644)) < 0) {

perror("make\_disk: cannot open file");

return -1;

}

memset(buf, 0, BLOCK\_SIZE);

for (cnt = 0; cnt < DISK\_BLOCKS; ++cnt)

write(f, buf, BLOCK\_SIZE);

close(f);

return 0;

}

int open\_disk(char \*name)

{

int f;

if (!name) {

fprintf(stderr, "open\_disk: invalid file name\n");

return -1;

}

if (active) {

fprintf(stderr, "open\_disk: disk is already open\n");

return -1;

}

if ((f = open(name, O\_RDWR, 0644)) < 0) {

perror("open\_disk: cannot open file");

return -1;

}

handle = f;

active = 1;

return 0;

}

int close\_disk()

{

if (!active) {

fprintf(stderr, "close\_disk: no open disk\n");

return -1;

}

close(handle);

active = handle = 0;

return 0;

}

int block\_write(int block, char \*buf)

{

if (!active) {

fprintf(stderr, "block\_write: disk not active\n");

return -1;

}

if ((block < 0) || (block >= DISK\_BLOCKS)) {

fprintf(stderr, "block\_write: block index out of bounds\n");

return -1;

}

if (lseek(handle, block \* BLOCK\_SIZE, SEEK\_SET) < 0) {

perror("block\_write: failed to lseek");

return -1;

}

if (write(handle, buf, BLOCK\_SIZE) < 0) {

perror("block\_write: failed to write");

return -1;

}

return 0;

}

int block\_read(int block, char \*buf)

{

if (!active) {

fprintf(stderr, "block\_read: disk not active\n");

return -1;

}

if ((block < 0) || (block >= DISK\_BLOCKS)) {

fprintf(stderr, "block\_read: block index out of bounds\n");

return -1;

}

if (lseek(handle, block \* BLOCK\_SIZE, SEEK\_SET) < 0) {

perror("block\_read: failed to lseek");

return -1;

}

if (read(handle, buf, BLOCK\_SIZE) < 0) {

perror("block\_read: failed to read");

return -1;

}

return 0;

}

Disk.h

#ifndef \_DISK\_H\_

#define \_DISK\_H\_

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define DISK\_BLOCKS 8192 /\* number of blocks on the disk \*/

#define BLOCK\_SIZE 4096 /\* block size on "disk" \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int make\_disk(char \*name); /\* create an empty, virtual disk file \*/

int open\_disk(char \*name); /\* open a virtual disk (file) \*/

int close\_disk(); /\* close a previously opened disk (file) \*/

int block\_write(int block, char \*buf);

/\* write a block of size BLOCK\_SIZE to disk \*/

int block\_read(int block, char \*buf);

/\* read a block of size BLOCK\_SIZE from disk \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#endif

Fs.c

//

// Created by Ethan Lee on 2016/11/25.

//

#include <stddef.h>

#include <printf.h>

#include <memory.h>

#include <stdlib.h>

#include "fs.h"

#include "disk.h"

super\_block\* super\_block\_ptr; // Super block

file\_info\* dir\_info; // Directory info

file\_descriptor fd\_table[MAX\_FILE\_DESCRIPTOR]; // File descriptor table

int make\_fs(char \*disk\_name)

{

make\_disk(disk\_name);

open\_disk(disk\_name);

/\* Initialize the super block \*/

super\_block\_ptr = (super\_block\*)malloc(sizeof(super\_block));

if (super\_block\_ptr == NULL) return -1;

super\_block\_ptr->dir\_index = 1;

super\_block\_ptr->dir\_len = 0;

super\_block\_ptr->data\_index = 2;

/\* write the super block to disk (block 0) \*/

char buf[BLOCK\_SIZE] = "";

// memset(void \*b, int c, size\_t len):

// writes len bytes of value c (converted to an unsigned char) to the string b.

memset(buf, 0, BLOCK\_SIZE);

// memcpy(void \*restrict dst, const void \*restrict src, size\_t n);

// copies n bytes from memory area src to memory area dst.

memcpy(buf, &super\_block\_ptr, sizeof(super\_block));

// write super block to disk

block\_write(0, buf);

free(super\_block\_ptr);

close\_disk();

printf("make\_fs()\t called successfully.\n");

return 0;

}

int mount\_fs(char \*disk\_name)

{

if(disk\_name == NULL) return -1;

open\_disk(disk\_name);

/\* read super block \*/

char buf[BLOCK\_SIZE] = "";

memset(buf, 0, BLOCK\_SIZE);

// read the first block data from disk

block\_read(0, buf);

memcpy(&super\_block\_ptr, buf, sizeof(super\_block\_ptr));

/\* read directory info \*/

dir\_info = (file\_info\*)malloc(BLOCK\_SIZE);

memset(buf, 0, BLOCK\_SIZE);

block\_read(super\_block\_ptr->dir\_index, buf);

memcpy(dir\_info, buf, sizeof(file\_info) \* super\_block\_ptr->dir\_len);

/\* clear file descriptors \*/

int i;

for(i = 0; i < MAX\_FILE\_DESCRIPTOR; ++i) {

fd\_table[i].used = false;

}

printf("mount\_fs()\t called successfully: file system [%s] mounted.\n", disk\_name);

return 0;

}

int umount\_fs(char \*disk\_name)

{

if(disk\_name == NULL) return -1;

/\* write directory info \*/

int i, j = 0;

file\_info\* file\_ptr = (file\_info\*)dir\_info;

char buf[BLOCK\_SIZE];

memset(buf, 0, BLOCK\_SIZE);

char\* block\_ptr = buf;

// backup dir\_info

for (i = 0; i < MAX\_FILE; ++i) {

if(dir\_info[i].used == true) {

memcpy(block\_ptr, &dir\_info[i], sizeof(dir\_info[i]));

block\_ptr += sizeof(file\_info);

}

}

// write back to disk

block\_write(super\_block\_ptr->dir\_index, buf);

/\* clear file descriptors \*/

for(j = 0; j < MAX\_FILE\_DESCRIPTOR; ++j) {

if(fd\_table[j].used == 1) {

fd\_table[j].used = false;

fd\_table[j].file = -1;

fd\_table[j].offset = 0;

}

}

free(dir\_info);

close\_disk();

printf("umount\_fs()\t called successfully: file system [%s] umounted.\n", disk\_name);

return 0;

}

int fs\_open(char \*name)

{

// try to find file

char file\_index = find\_file(name);

if(file\_index < 0) { // file not found

fprintf(stderr, "fs\_open()\t error: file [%s] does not exist.\n",name);

return -1;

}

// is fd avaliable(not used)

int fd = find\_free\_file\_des(file\_index);

if (fd < 0){

fprintf(stderr, "fs\_open()\t error: exceed the maximum file descriptor number.\n");

return -1;

}

dir\_info[file\_index].fd\_count++;

printf("fs\_open()\t called successfully: file [%s] opened.\n", name);

return fd;

}

int fs\_close(int fildes)

{

if(fildes < 0 || fildes >= MAX\_FILE\_DESCRIPTOR || !fd\_table[fildes].used) {

return -1;

}

file\_descriptor\* fd = &fd\_table[fildes];

dir\_info[fd->file].fd\_count--;

fd->used = false;

printf("fs\_close()\t called successfully: file [%s] closed\n", dir\_info[fd->file].name);

return 0;

}

int fs\_create(char \*name)

{

char file\_index = find\_file(name);

if (file\_index < 0){ // Create file

char i;

for(i = 0; i < MAX\_FILE; i++) {

// find a dir\_info that not used for the new file

if(dir\_info[i].used == false) {

super\_block\_ptr->dir\_len++;

/\* Initialize file information \*/

dir\_info[i].used = true;

strcpy(dir\_info[i].name, name);

dir\_info[i].size = 0;

dir\_info[i].head = -1;

dir\_info[i].num\_blocks = 0;

dir\_info[i].fd\_count = 0;

printf("fs\_create()\t called successfully: file [%s] created.\n", name);

return 0;

}

}

fprintf(stderr, "fs\_create()\t error: exceed the maximum file number.\n");

return -1;

} else { // File already exists

fprintf(stderr, "fs\_create()\t error: file [%s] already exists\n",name);

return 0;

}

}

int fs\_delete(char \*name)

{

char i;

for(i = 0; i < MAX\_FILE; ++i) {

if(strcmp(dir\_info[i].name, name) == 0) {

char file\_index = i;

file\_info\* file = &dir\_info[i];

int block\_index = file->head;

int block\_count = file->num\_blocks;

if(dir\_info[i].fd\_count != 0) { // File is currently open

fprintf(stderr, "fs\_delete()\t error: file [%s] is currently open.\n",name);

return -1;

}

// Remove file information

super\_block\_ptr->dir\_len--;

file->used = false;

strcpy(file->name, "");

file->size = 0;

file->fd\_count = 0;

/\* Free file blocks \*/

char buf1[BLOCK\_SIZE] = "";

char buf2[BLOCK\_SIZE] = "";

block\_read(super\_block\_ptr->data\_index, buf1);

block\_read(super\_block\_ptr->data\_index + 1, buf2);

while (block\_count > 0){

if (block\_index < BLOCK\_SIZE){

buf1[block\_index] = '\0';

} else {

buf2[block\_index - BLOCK\_SIZE] = '\0';

}

block\_index = find\_next\_block(file->head, file\_index);

block\_count--;

}

dir\_info[i].head = -1;

dir\_info[i].num\_blocks = 0;

block\_write(super\_block\_ptr->data\_index, buf1);

block\_write(super\_block\_ptr->data\_index + 1, buf2);

printf("fs\_delete()\t called successfully: file [%s] deleted.\n", name);

return 0;

}

}

fprintf(stderr, "fs\_delete()\t error: file [%s] does not exists\n", name);

return -1;

}

int fs\_read(int fildes, void \*buf, size\_t nbyte)

{

if(nbyte <= 0 || !fd\_table[fildes].used) {

return -1;

}

int i, j = 0;

char \*dst = buf;

char block[BLOCK\_SIZE] = "";

char file\_index = fd\_table[fildes].file;

file\_info\* file = &dir\_info[file\_index];

int block\_index = file->head;

int block\_count = 0;

int offset = fd\_table[fildes].offset;

/\* load current block \*/

while (offset >= BLOCK\_SIZE){

block\_index = find\_next\_block(block\_index, file\_index);

block\_count++;

offset -= BLOCK\_SIZE;

}

block\_read(block\_index, block);

/\* read current block \*/

int read\_count = 0;

for(i = offset; i < BLOCK\_SIZE; i++) {

dst[read\_count++] = block[i];

if(read\_count == (int)nbyte) {

fd\_table[fildes].offset += read\_count;

return read\_count;

}

}

block\_count++;

/\* read the following blocks \*/

strcpy(block,"");

while(read\_count < (int)nbyte && block\_count <= file->num\_blocks) {

block\_index = find\_next\_block(block\_index, file\_index);

strcpy(block,"");

block\_read(block\_index, block);

for(j=0; j < BLOCK\_SIZE; j++, i++) {

dst[read\_count++] = block[j];

if(read\_count == (int)nbyte ) {

fd\_table[fildes].offset += read\_count;

return read\_count;

}

}

block\_count++;

}

fd\_table[fildes].offset += read\_count;

return read\_count;

}

int fs\_write(int fildes, void \*buf, size\_t nbyte)

{

if(nbyte <= 0 || !fd\_table[fildes].used) {

return -1;

}

int i = 0;

char \*src = buf;

char block[BLOCK\_SIZE] = "";

char file\_index = fd\_table[fildes].file;

file\_info\* file = &dir\_info[file\_index];

int block\_index = file->head;

int size = file->size;

int block\_count = 0;

int offset = fd\_table[fildes].offset;

/\* load current block \*/

while (offset >= BLOCK\_SIZE){

block\_index = find\_next\_block(block\_index, file\_index);

block\_count++;

offset -= BLOCK\_SIZE;

}

int write\_count = 0;

if (block\_index != -1){

/\* write current block \*/

block\_read(block\_index, block);

for(i = offset; i < BLOCK\_SIZE; i++) {

block[i] = src[write\_count++];

if (write\_count == (int)nbyte || write\_count == strlen(src)) {

block\_write(block\_index, block);

fd\_table[fildes].offset += write\_count;

if(size < fd\_table[fildes].offset){

file->size = fd\_table[fildes].offset;

}

return write\_count;

}

}

block\_write(block\_index, block);

block\_count++;

}

/\* write the allocated blocks \*/

strcpy(block, "");

while(write\_count < (int)nbyte && write\_count < strlen(src) && block\_count < file->num\_blocks) {

block\_index = find\_next\_block(block\_index, file\_index);

for(i = 0; i < BLOCK\_SIZE; i++) {

block[i] = src[write\_count++];

if(write\_count == (int)nbyte || write\_count == strlen(src)) {

block\_write(block\_index, block);

fd\_table[fildes].offset += write\_count;

if(size < fd\_table[fildes].offset){

file->size = fd\_table[fildes].offset;

}

return write\_count;

}

}

block\_write(block\_index, block);

block\_count++;

}

/\* write into new blocks \*/

strcpy(block, "");

while(write\_count < (int)nbyte && write\_count < strlen(src)) {

block\_index = find\_free\_block(file\_index);

file->num\_blocks++;

if (file->head == -1){

file->head = block\_index;

}

if (block\_index < 0){

fprintf(stderr, "fs\_write()\t error: No free blocks.\n");

return -1;

}

for(i = 0; i < BLOCK\_SIZE; i++) {

block[i] = src[write\_count++];

if(write\_count == (int)nbyte || write\_count == strlen(src)) {

block\_write(block\_index, block);

fd\_table[fildes].offset += write\_count;

if(size < fd\_table[fildes].offset){

file->size = fd\_table[fildes].offset;

}

return write\_count;

}

}

block\_write(block\_index, block);

}

fd\_table[fildes].offset += write\_count;

if(size < fd\_table[fildes].offset){

file->size = fd\_table[fildes].offset;

}

return write\_count;

}

int fs\_get\_filesize(int fildes){

if(!fd\_table[fildes].used){

fprintf(stderr, "fs\_get\_filesize()\t error: Invalid file descriptor.\n");

return -1;

}

return dir\_info[fd\_table[fildes].file].size;

}

int fs\_lseek(int fildes, off\_t offset)

{

if (offset > dir\_info[fd\_table[fildes].file].size || offset < 0){

fprintf(stderr, "fs\_lseek()\t error: Can't set the file pointer beyond the file range.\n");

return -1;

} else if(!fd\_table[fildes].used){

fprintf(stderr, "fs\_lseek()\t error: Invalid file descriptor.\n");

return -1;

} else {

fd\_table[fildes].offset = (int)offset;

printf("fs\_lseek()\t called successfully.\n");

return 0;

}

}

int fs\_truncate(int fildes, off\_t length)

{

char file\_index = fd\_table[fildes].file;

file\_info\* file = &dir\_info[file\_index];

if(!fd\_table[fildes].used){

fprintf(stderr, "fs\_truncate()\t error: Invalid file descriptor.\n");

return -1;

}

if (length > file->size || length < 0) {

fprintf(stderr, "fs\_truncate()\t error: Can't set the offset beyond the file range.\n");

return -1;

}

/\* free blocks \*/

int new\_block\_num = (int) (length + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

int i;

int block\_index = file->head;

for (i = 0; i < new\_block\_num; ++i) {

block\_index = find\_next\_block(block\_index, file\_index);

}

while (block\_index > 0){

char buf[BLOCK\_SIZE] = "";

if (block\_index < BLOCK\_SIZE){

block\_read(super\_block\_ptr->data\_index, buf);

buf[block\_index] = '\0';

block\_write(super\_block\_ptr->data\_index, buf);

} else {

block\_read(super\_block\_ptr->data\_index + 1, buf);

buf[block\_index - BLOCK\_SIZE] = '\0';

block\_write(super\_block\_ptr->data\_index + 1, buf);

}

block\_index = find\_next\_block(block\_index, file\_index);

}

/\* modify file information \*/

file->size = (int)length;

file->num\_blocks = new\_block\_num;

/\* truncate fd offset \*/

for(i = 0; i < MAX\_FILE\_DESCRIPTOR; i++) {

if(fd\_table[i].used == true && fd\_table[i].file == file\_index) {

fd\_table[i].offset = (int)length;

}

}

printf("fs\_truncate()called successfully.\n");

return 0;

}

char find\_file(char\* name)

{

char i;

for(i = 0; i < MAX\_FILE; i++) {

if(dir\_info[i].used == 1 && strcmp(dir\_info[i].name, name) == 0) {

return i; // return the file index

}

}

return -1; // file not found

}

int find\_free\_file\_des(char file\_index)

{

int i;

for(i = 0; i < MAX\_FILE\_DESCRIPTOR; i++) {

if(fd\_table[i].used == false) {

fd\_table[i].used = true;

fd\_table[i].file = file\_index;

fd\_table[i].offset = 0;

return i; // return the file descriptor number

}

}

fprintf(stderr, "find\_free\_file\_des()\t error: no available file descriptor.\n");

return -1; // no empty file descriptor available

}

int find\_free\_block(char file\_index)

{

char buf1[BLOCK\_SIZE] = "";

char buf2[BLOCK\_SIZE] = "";

block\_read(super\_block\_ptr->data\_index, buf1);

block\_read(super\_block\_ptr->data\_index + 1, buf2);

int i;

for(i = 4; i < BLOCK\_SIZE; i++) {

if(buf1[i] == '\0') {

buf1[i] = (char)(file\_index + 1);

block\_write(super\_block\_ptr->data\_index, buf1);

return i; // return block number

}

}

for(i = 0; i < BLOCK\_SIZE; i++) {

if(buf2[i] == '\0') {

buf2[i] = (char)(file\_index + 1);

block\_write(super\_block\_ptr->data\_index, buf2);

return i; // return block number

}

}

fprintf(stderr, "find\_free\_block()\t error: no available blocks.\n");

return -1; // no free blocks

}

int find\_next\_block(int current, char file\_index){

char buf[BLOCK\_SIZE] = "";

int i;

if (current < BLOCK\_SIZE){

block\_read(super\_block\_ptr->data\_index, buf);

for(i = current + 1; i < BLOCK\_SIZE; i++) {

if (buf[i] == (file\_index + 1)){

return i;

}

}

} else {

block\_read(super\_block\_ptr->data\_index + 1, buf);

for(i = current - BLOCK\_SIZE + 1; i < BLOCK\_SIZE; i++) {

if (buf[i] == (file\_index + 1)){

return i + BLOCK\_SIZE;

}

}

}

return -1; // no next block

}

int main()

{

int i;

char\* disk\_name = "RootDir";

if(make\_fs(disk\_name) < 0) {

fprintf(stderr, "make\_fs()\t error.\n");

}

if(mount\_fs(disk\_name) < 0) {

fprintf(stderr, "mount\_fs()\t error.\n");

}

if(fs\_create("test.txt") < 0) {

fprintf(stderr, "fs\_create()\t error.\n");

}

/\* fs\_delete() test \*/

if(fs\_delete("test.txt") < 0) {

fprintf(stderr, "fs\_delete()\t error.\n");

}

if(fs\_create("test.txt") < 0) {

fprintf(stderr, "fs\_create()\t error.\n");

}

/\* fs\_write() test \*/

int fd1;

if((fd1 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

int fd2;

if((fd2 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

char str1[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str1[i] = 'a';

str1[i + BLOCK\_SIZE / 2] = 'b';

str1[i + BLOCK\_SIZE] = 'c';

str1[i + BLOCK\_SIZE \* 3 / 2] = 'd';

}

char str2[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str2[i] = 'e';

str2[i + BLOCK\_SIZE / 2] = 'f';

str2[i + BLOCK\_SIZE] = 'g';

str2[i + BLOCK\_SIZE \* 3 / 2] = 'h';

}

fs\_write(fd1, str1, BLOCK\_SIZE \* 2);

fs\_write(fd1, str2, BLOCK\_SIZE \* 2);

/\* fs\_lseek() test \*/

char str3[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str3[i] = 'i';

str3[i + BLOCK\_SIZE / 2] = 'j';

str3[i + BLOCK\_SIZE] = 'k';

str3[i + BLOCK\_SIZE \* 3 / 2] = 'l';

}

fs\_lseek(fd2, BLOCK\_SIZE \* 2);

fs\_write(fd2, str3, BLOCK\_SIZE \* 2);

/\* fs\_truncate() test \*/

fs\_truncate(fd2, BLOCK\_SIZE \* 5 / 2);

char str4[BLOCK\_SIZE \* 3 / 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str4[i] = 'm';

str4[i + BLOCK\_SIZE / 2] = 'n';

str4[i + BLOCK\_SIZE] = 'o';

}

fs\_write(fd2, str4, BLOCK\_SIZE \* 3 / 2);

if(fs\_close(fd1) < 0) {

fprintf(stderr, "fs\_close()\t error.\n");

}

if(fs\_close(fd2) < 0) {

fprintf(stderr, "fs\_close()\t error.\n");

}

if(umount\_fs(disk\_name) < 0) {

fprintf(stderr, "umount\_fs()\t error.\n");

}

if(mount\_fs(disk\_name) < 0) {

fprintf(stderr, "mount\_fs()\t error.\n");

}

if((fd1 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

/\* fs\_read() Test \*/

char buf1[BLOCK\_SIZE \* 4] = "";

char val1[BLOCK\_SIZE \* 4];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

val1[i] = 'a';

val1[i + BLOCK\_SIZE / 2] = 'b';

val1[i + BLOCK\_SIZE] = 'c';

val1[i + BLOCK\_SIZE \* 3 / 2] = 'd';

val1[i + BLOCK\_SIZE \* 2] = 'i';

val1[i + BLOCK\_SIZE \* 5 / 2] = 'm';

val1[i + BLOCK\_SIZE \* 3] = 'n';

val1[i + BLOCK\_SIZE \* 7 / 2] = 'o';

}

int fd3;

if((fd3 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

if(fs\_read(fd3, buf1, BLOCK\_SIZE \* 4) < 0) {

fprintf(stderr, "fs\_read()\t error.\n");

} else {

printf("fs\_read()\t called successfully.\n");

}

for (i = 0; i < BLOCK\_SIZE \* 4; ++i) {

if (buf1[i] != val1[i]) {

fprintf(stderr, "fs\_read()\t error: Content [%d] error: [%c] and [%c].\n", i, buf1[i], val1[i]);

break;

}

}

/\* fs\_get\_filesize() Test \*/

int file\_size;

if((file\_size = fs\_get\_filesize(fd1)) < 0) {

fprintf(stderr, "fs\_get\_filesize()\t error.\n");

} else {

printf("fs\_get\_filesize()\t called successfully: The file size is %d\n", file\_size);

}

/\* multiple file test \*/

char j;

int k;

for (j = 0; j < 63; ++j) {

/\* create \*/

char\* file\_name = (char\*)malloc(9);

char index[2];

index[0] = (char) (j + 48);

index[1] = '\0';

strcat(file\_name, "test");

strcat(file\_name, index);

strcat(file\_name, ".txt");

fs\_create(file\_name);

/\* write \*/

char str[BLOCK\_SIZE \* 64];

int fd\_w = fs\_open(file\_name);

for(k = 0; k < BLOCK\_SIZE \* 64; ++k) {

str[k] = (char) (j + 48);

}

fs\_write(fd\_w, str, BLOCK\_SIZE \* 64);

/\* read \*/

char buf[BLOCK\_SIZE \* 64];

int fd\_r = fs\_open(file\_name);

fs\_read(fd\_r, buf, BLOCK\_SIZE \* 64);

for (k = 0; k < BLOCK\_SIZE \* 64; ++k) {

if (str[k] != buf[k]) {

fprintf(stderr, "fs\_read()\t error: Content [%d] error: [%c] and [%c].\n", k, str[k], buf[k]);

break;

}

}

/\* close \*/

fs\_close(fd\_w);

fs\_close(fd\_r);

memset(file\_name, 0, 9);

free(file\_name);

}

if(umount\_fs(disk\_name) < 0) {

fprintf(stderr, "umount\_fs()\t error.\n");

}

return 0;

};

Fs.h

//

// Created by Ethan Lee on 2016/11/25.

//

#ifndef SIMPLEFILESYSTEM\_FS\_H

#define SIMPLEFILESYSTEM\_FS\_H

#define MAX\_FILENAME\_LEN 15

#define MAX\_FILE\_DESCRIPTOR 32

#define MAX\_FILE 64

typedef enum { false, true } bool;

typedef struct

{

// Directory info

int dir\_index;

int dir\_len;

// Data info

int data\_index;

} super\_block;

/\* file information \*/

typedef struct

{

bool used; // whether the file is being used

char name[MAX\_FILENAME\_LEN]; // file name

int size; // file size

int head; // first data block

int num\_blocks; // number of blocks

int fd\_count; // number of file descriptors using this file

} file\_info;

/\* file descriptor \*/

typedef struct

{

bool used; // whether the file descriptor is being used

char file; // file index

int offset; // read offset used by fs\_read()

} file\_descriptor;

int make\_fs(char \*disk\_name);

int mount\_fs(char \*disk\_name);

int umount\_fs(char \*disk\_name);

int fs\_open(char \*name);

int fs\_close(int fildes);

int fs\_create(char \*name);

int fs\_delete(char \*name);

int fs\_read(int fildes, void \*buf, size\_t nbyte);

int fs\_write(int fildes, void \*buf, size\_t nbyte);

int fs\_get\_filesize(int fildes);

int fs\_lseek(int fildes, off\_t offset);

int fs\_truncate(int fildes, off\_t length);

/\* Helper function \*/

char find\_file(char\* name);

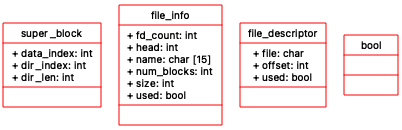
int find\_free\_file\_des(char file\_index);

int find\_free\_block(char file\_index);

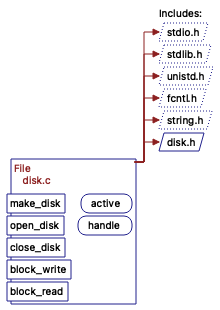
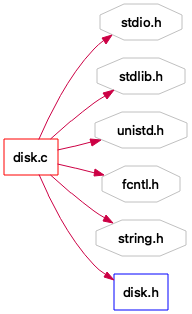
int find\_next\_block(int current, char file\_index);

#endif //SIMPLEFILESYSTEM\_FS\_H

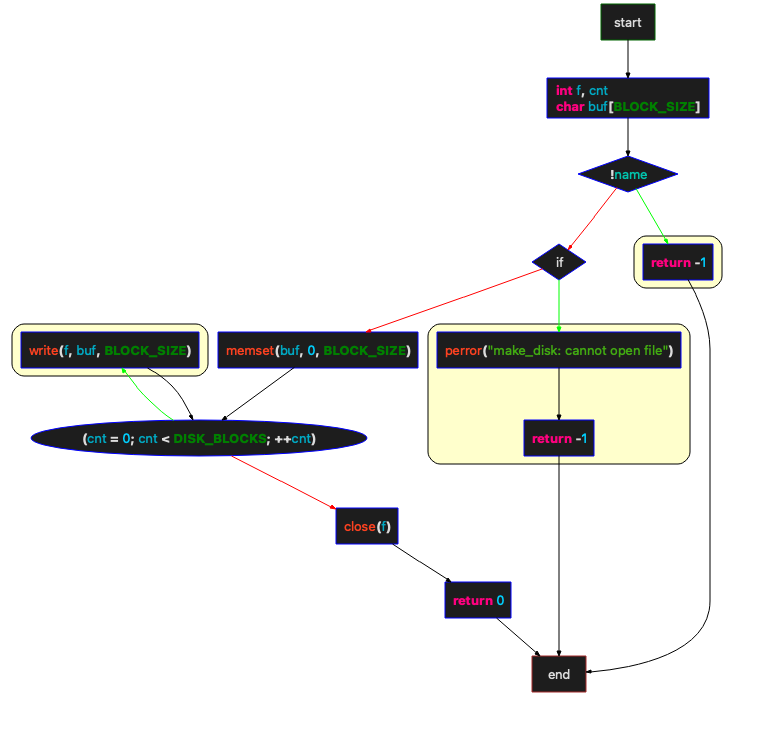
### 程序流程图



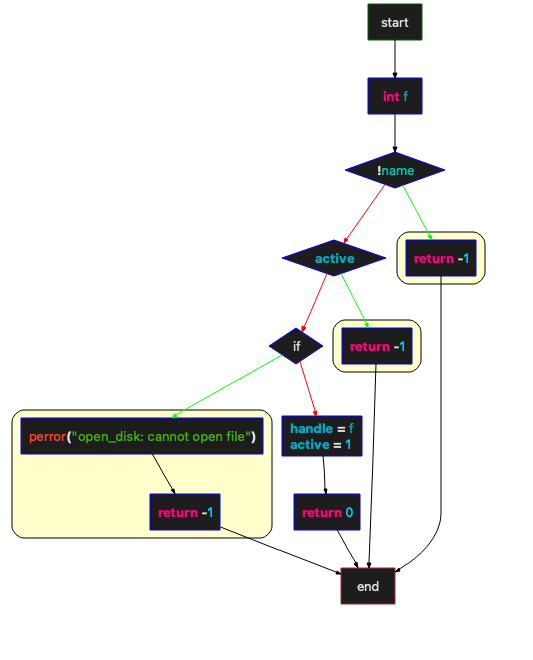
据结构

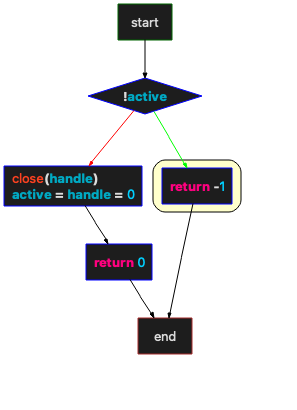
Disk.c头文件依赖情况



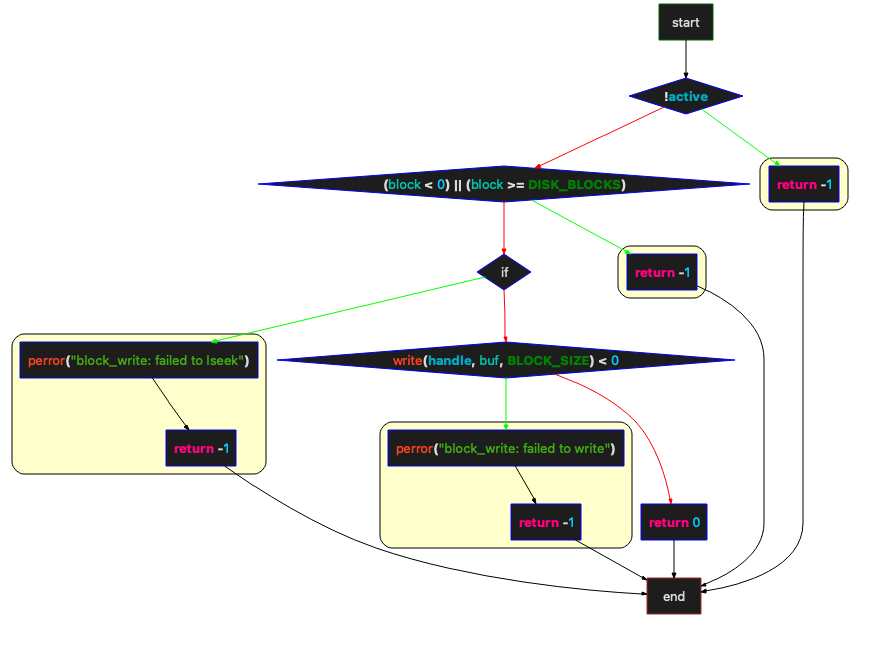
Make\_disk函数流程图



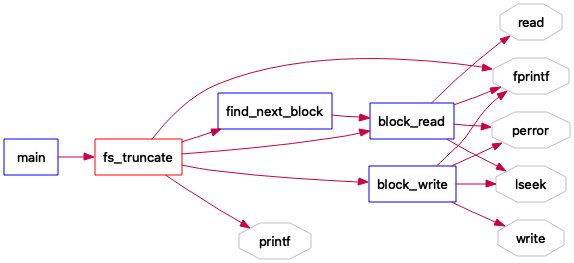
open\_disk函数



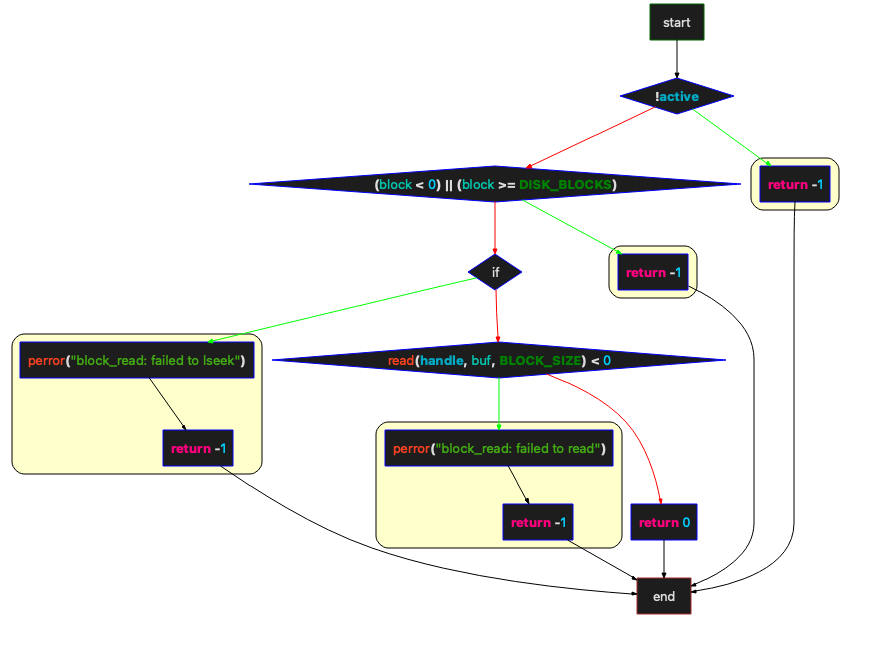
Close\_disk函数流程图



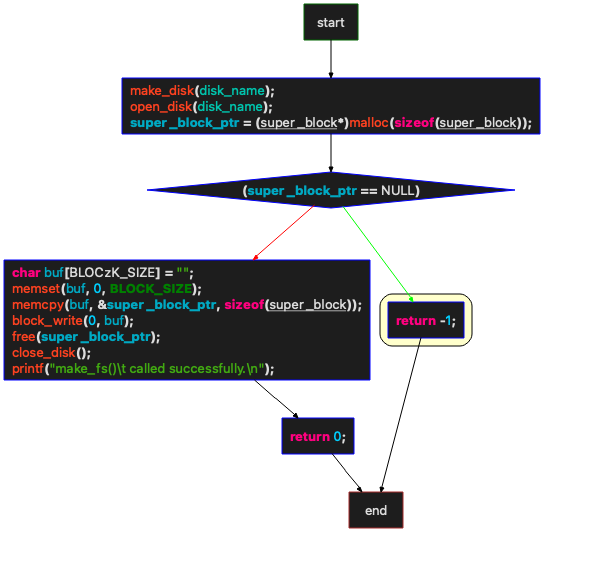
Block\_write函数流程图



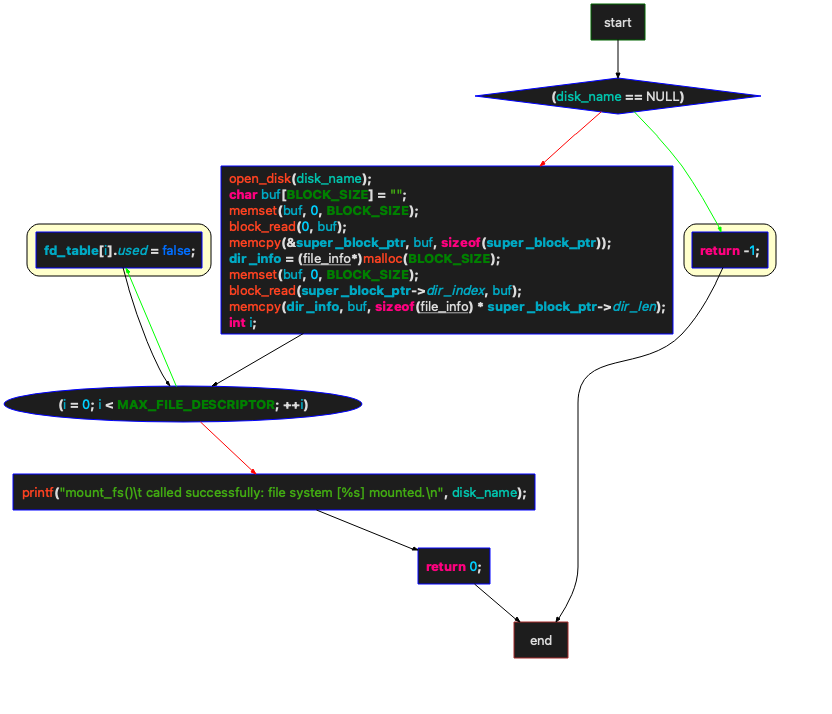
函数调用关系图



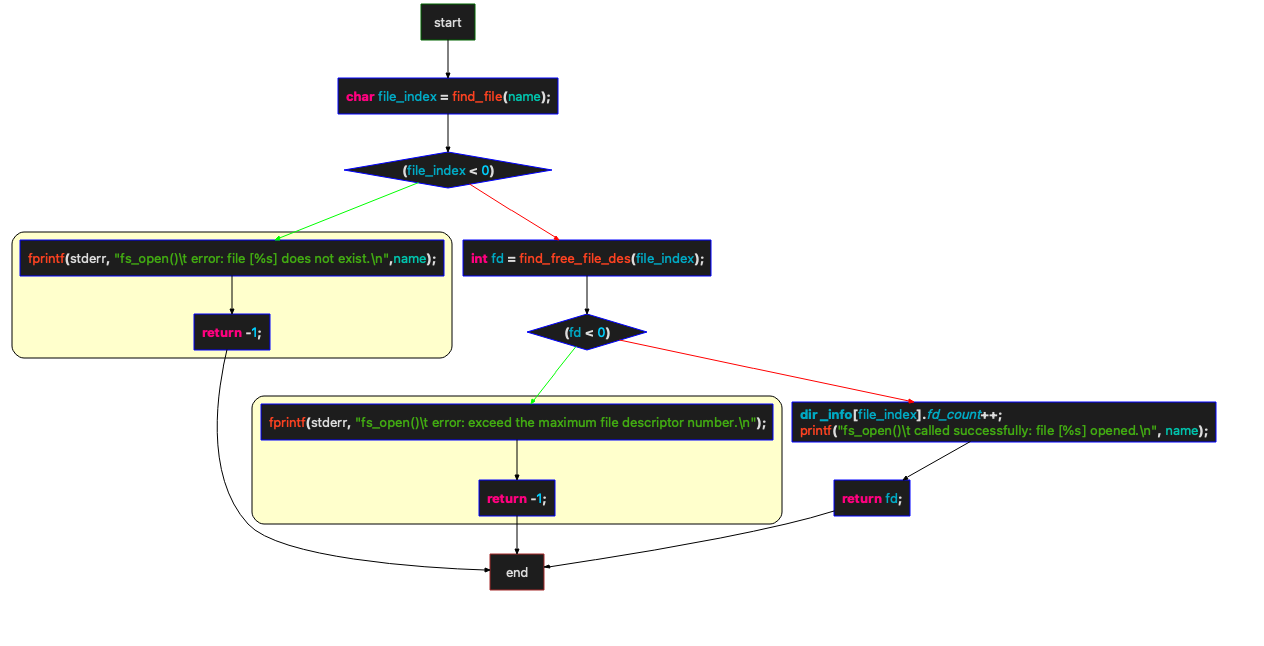
block\_read函数流程图



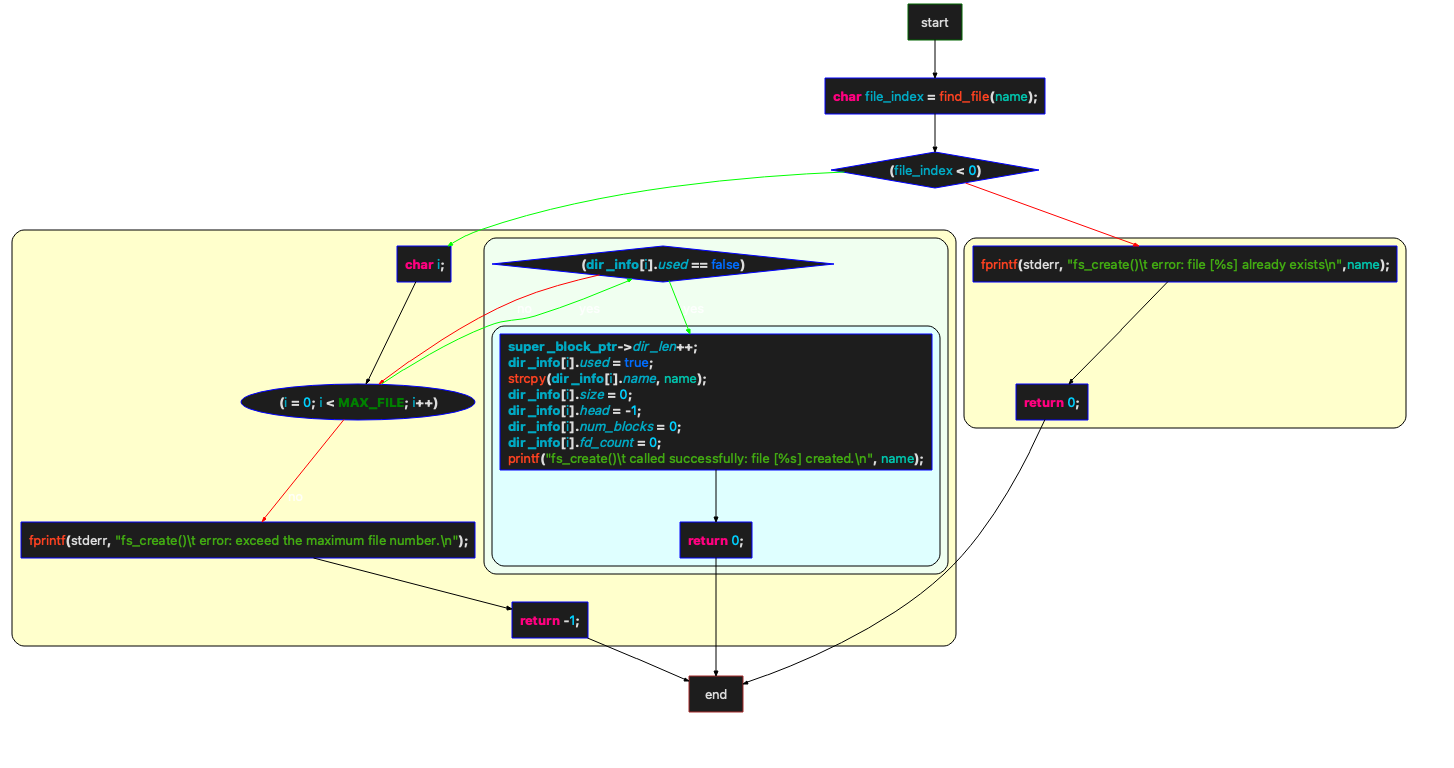
Make\_fs函数流程图



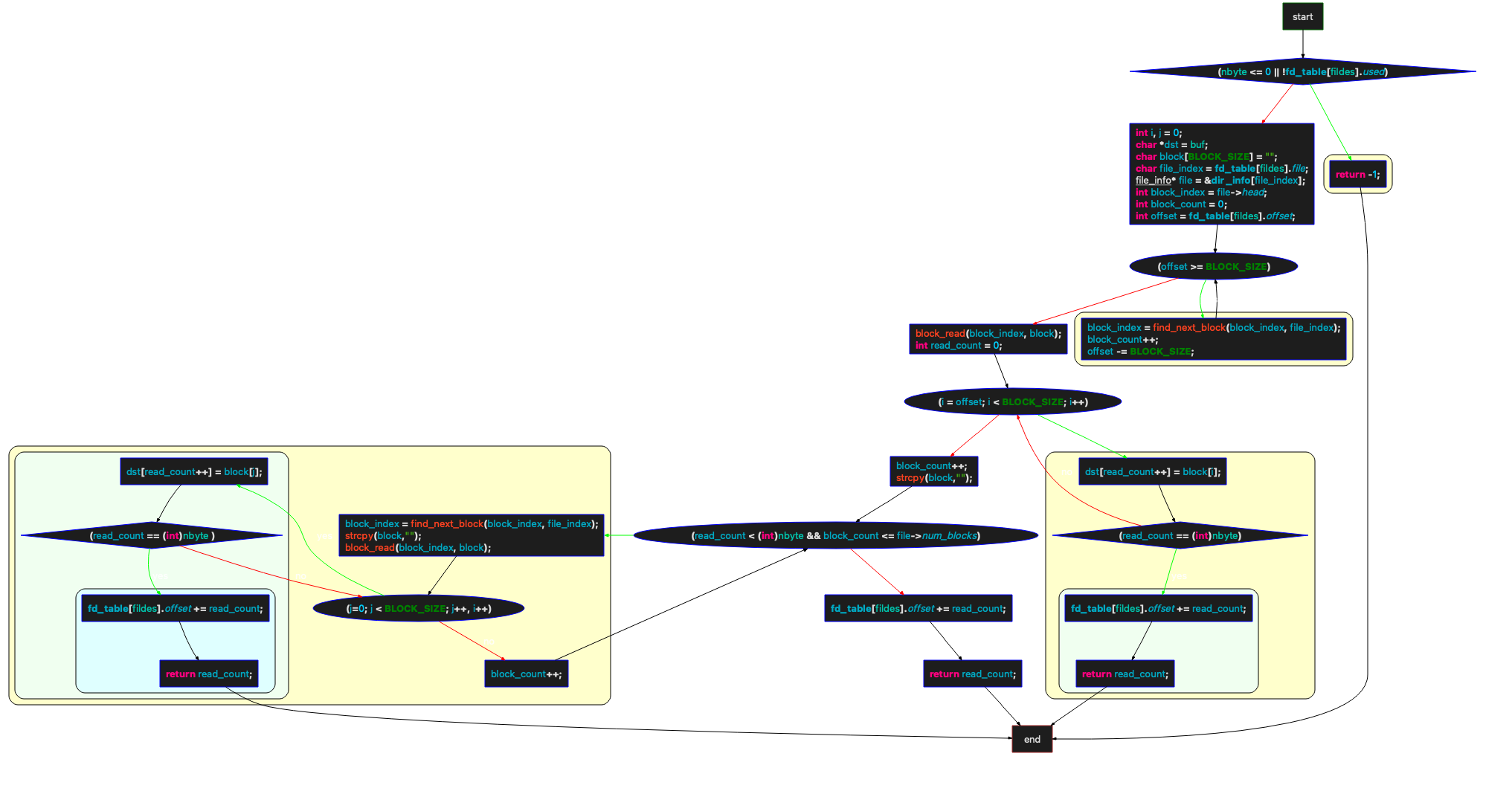
Mount\_fs函数流程图



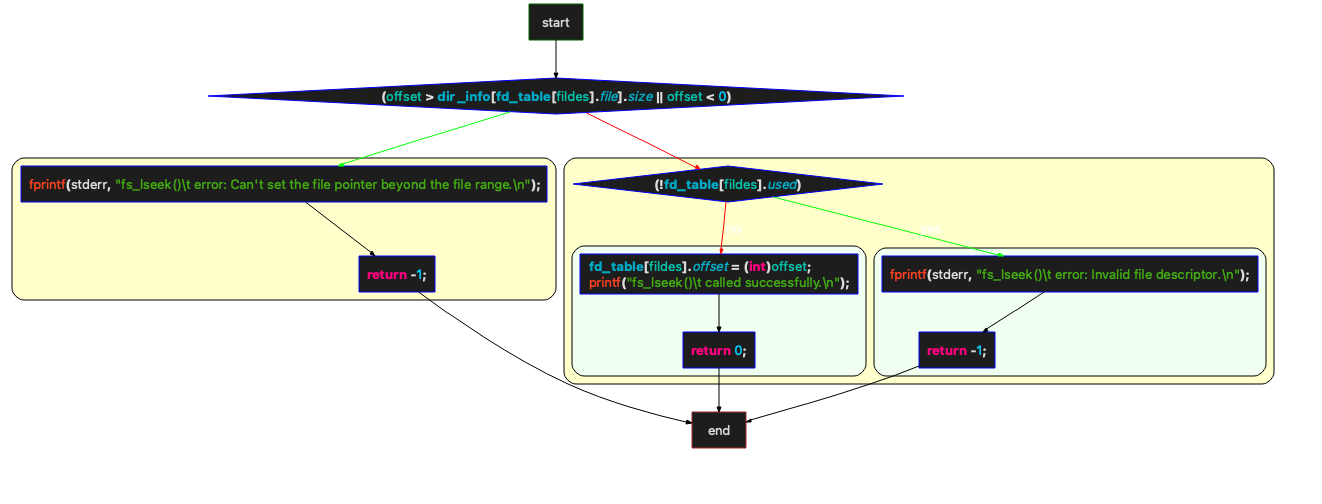
Fs\_open函数流程图



Fs\_create



Fs\_read



Fs\_seek函数流程图

## 项目展示

### 测试代码

int main()

{

int i;

char\* disk\_name = "RootDir";

if(make\_fs(disk\_name) < 0) {

fprintf(stderr, "make\_fs()\t error.\n");

}

if(mount\_fs(disk\_name) < 0) {

fprintf(stderr, "mount\_fs()\t error.\n");

}

if(fs\_create("test.txt") < 0) {

fprintf(stderr, "fs\_create()\t error.\n");

}

/\* fs\_delete() test \*/

if(fs\_delete("test.txt") < 0) {

fprintf(stderr, "fs\_delete()\t error.\n");

}

if(fs\_create("test.txt") < 0) {

fprintf(stderr, "fs\_create()\t error.\n");

}

/\* fs\_write() test \*/

int fd1;

if((fd1 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

int fd2;

if((fd2 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

char str1[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str1[i] = 'a';

str1[i + BLOCK\_SIZE / 2] = 'b';

str1[i + BLOCK\_SIZE] = 'c';

str1[i + BLOCK\_SIZE \* 3 / 2] = 'd';

}

char str2[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str2[i] = 'e';

str2[i + BLOCK\_SIZE / 2] = 'f';

str2[i + BLOCK\_SIZE] = 'g';

str2[i + BLOCK\_SIZE \* 3 / 2] = 'h';

}

fs\_write(fd1, str1, BLOCK\_SIZE \* 2);

fs\_write(fd1, str2, BLOCK\_SIZE \* 2);

/\* fs\_lseek() test \*/

char str3[BLOCK\_SIZE \* 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str3[i] = 'i';

str3[i + BLOCK\_SIZE / 2] = 'j';

str3[i + BLOCK\_SIZE] = 'k';

str3[i + BLOCK\_SIZE \* 3 / 2] = 'l';

}

fs\_lseek(fd2, BLOCK\_SIZE \* 2);

fs\_write(fd2, str3, BLOCK\_SIZE \* 2);

/\* fs\_truncate() test \*/

fs\_truncate(fd2, BLOCK\_SIZE \* 5 / 2);

char str4[BLOCK\_SIZE \* 3 / 2];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

str4[i] = 'm';

str4[i + BLOCK\_SIZE / 2] = 'n';

str4[i + BLOCK\_SIZE] = 'o';

}

fs\_write(fd2, str4, BLOCK\_SIZE \* 3 / 2);

if(fs\_close(fd1) < 0) {

fprintf(stderr, "fs\_close()\t error.\n");

}

if(fs\_close(fd2) < 0) {

fprintf(stderr, "fs\_close()\t error.\n");

}

if(umount\_fs(disk\_name) < 0) {

fprintf(stderr, "umount\_fs()\t error.\n");

}

if(mount\_fs(disk\_name) < 0) {

fprintf(stderr, "mount\_fs()\t error.\n");

}

if((fd1 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

/\* fs\_read() Test \*/

char buf1[BLOCK\_SIZE \* 4] = "";

char val1[BLOCK\_SIZE \* 4];

for(i = 0; i < BLOCK\_SIZE / 2; i++) {

val1[i] = 'a';

val1[i + BLOCK\_SIZE / 2] = 'b';

val1[i + BLOCK\_SIZE] = 'c';

val1[i + BLOCK\_SIZE \* 3 / 2] = 'd';

val1[i + BLOCK\_SIZE \* 2] = 'i';

val1[i + BLOCK\_SIZE \* 5 / 2] = 'm';

val1[i + BLOCK\_SIZE \* 3] = 'n';

val1[i + BLOCK\_SIZE \* 7 / 2] = 'o';

}

int fd3;

if((fd3 = fs\_open("test.txt")) < 0) {

fprintf(stderr, "fs\_open()\t error.\n");

}

if(fs\_read(fd3, buf1, BLOCK\_SIZE \* 4) < 0) {

fprintf(stderr, "fs\_read()\t error.\n");

} else {

printf("fs\_read()\t called successfully.\n");

}

for (i = 0; i < BLOCK\_SIZE \* 4; ++i) {

if (buf1[i] != val1[i]) {

fprintf(stderr, "fs\_read()\t error: Content [%d] error: [%c] and [%c].\n", i, buf1[i], val1[i]);

break;

}

}

/\* fs\_get\_filesize() Test \*/

int file\_size;

if((file\_size = fs\_get\_filesize(fd1)) < 0) {

fprintf(stderr, "fs\_get\_filesize()\t error.\n");

} else {

printf("fs\_get\_filesize()\t called successfully: The file size is %d\n", file\_size);

}

/\* multiple file test \*/

char j;

int k;

for (j = 0; j < 63; ++j) {

/\* create \*/

char\* file\_name = (char\*)malloc(9);

char index[2];

index[0] = (char) (j + 48);

index[1] = '\0';

strcat(file\_name, "test");

strcat(file\_name, index);

strcat(file\_name, ".txt");

fs\_create(file\_name);

/\* write \*/

char str[BLOCK\_SIZE \* 64];

int fd\_w = fs\_open(file\_name);

for(k = 0; k < BLOCK\_SIZE \* 64; ++k) {

str[k] = (char) (j + 48);

}

fs\_write(fd\_w, str, BLOCK\_SIZE \* 64);

/\* read \*/

char buf[BLOCK\_SIZE \* 64];

int fd\_r = fs\_open(file\_name);

fs\_read(fd\_r, buf, BLOCK\_SIZE \* 64);

for (k = 0; k < BLOCK\_SIZE \* 64; ++k) {

if (str[k] != buf[k]) {

fprintf(stderr, "fs\_read()\t error: Content [%d] error: [%c] and [%c].\n", k, str[k], buf[k]);

break;

}

}

/\* close \*/

fs\_close(fd\_w);

fs\_close(fd\_r);

memset(file\_name, 0, 9);

free(file\_name);

}

if(umount\_fs(disk\_name) < 0) {

fprintf(stderr, "umount\_fs()\t error.\n");

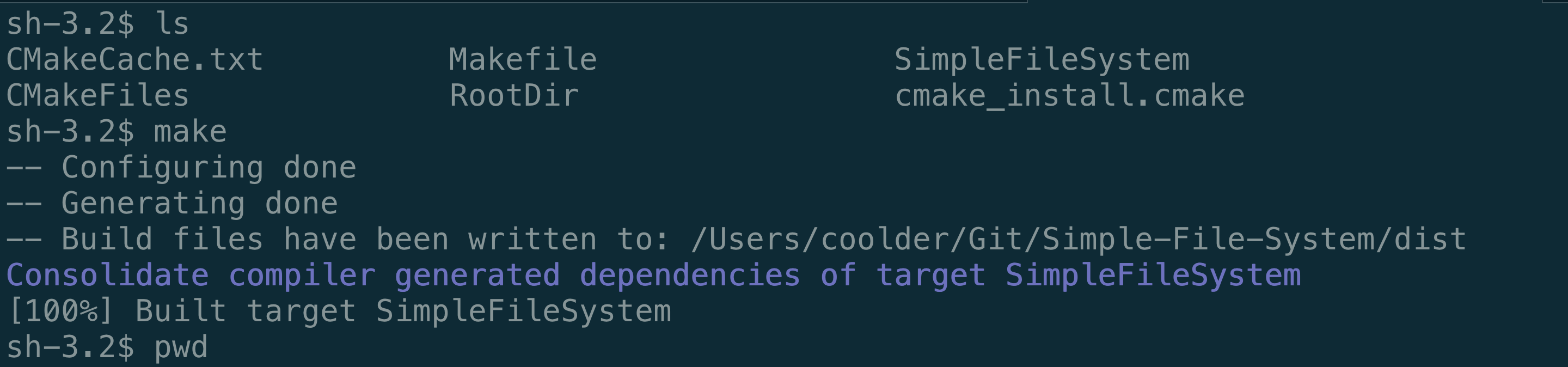
}

return 0;

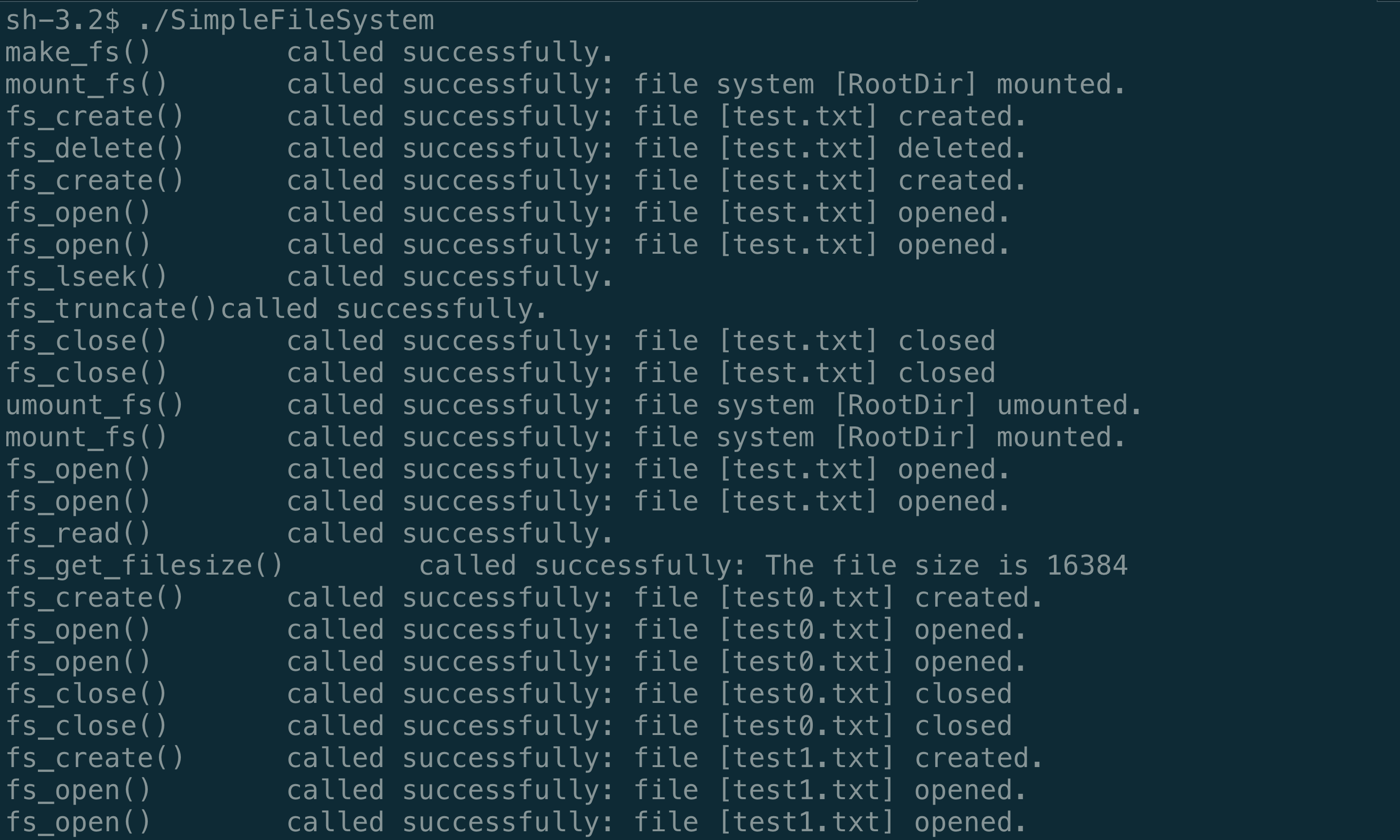
};

### 运行结果

使用 make进行编译



运行测试程序



## 项目总结

通过使用C语言编写简单的文件系统，掌握了C语言的基本知识和一些高级用法如指针的指针和指针数组的转换，结构体数组和结构体指针的应用。学习到了超级块的数据结构在C语言中的体现，文件如何被创建，删除等。使用malloc分配的内存，使用完后必须要释放内存，否则会导致内存泄漏。