# Linux Fat文件系统设计

## 开发环境

vim --version ─╯

VIM - Vi IMproved 8.2 (2019 Dec 12, compiled Apr 19 2022 21:42:29)

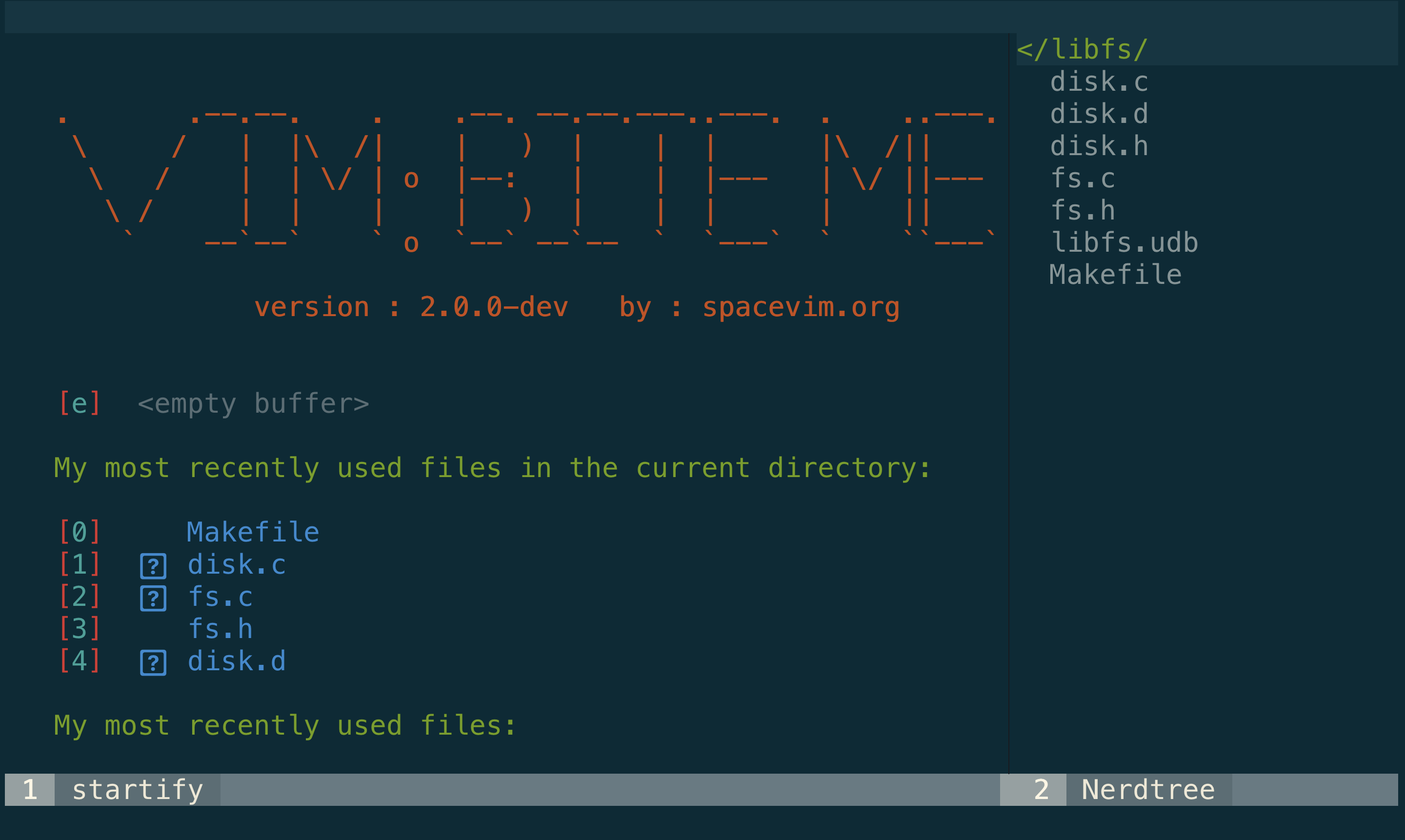
macOS version - arm64

Included patches: 1-4113

Compiled by root@apple.com

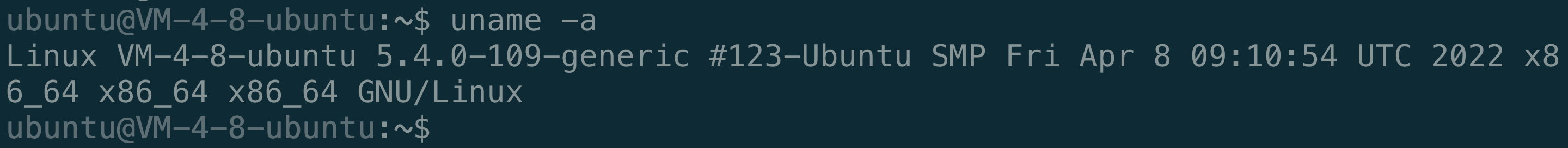
Normal version without GUI. Features included (+) or not (-):

Spacevim



## 实验环境

Uname -a

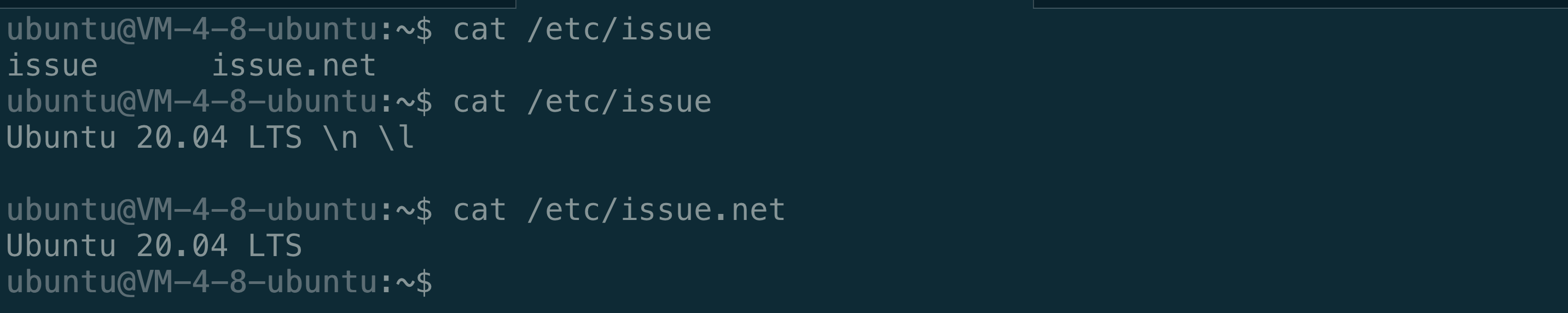


ubuntu@VM-4-8-ubuntu:~$ cat /etc/issue

issue issue.net

ubuntu@VM-4-8-ubuntu:~$ cat /etc/issue

Ubuntu 20.04 LTS \n \l



## 流程分析

### 文件系统数据结构

struct disk {

/\* File descriptor \*/

int fd;

/\* Block count \*/

size\_t bcount;

};

需要使用\_\_packed\_\_指令进行内存对其，gcc专有

struct superblock {

uint8\_t signature[8]; // ECS150FS

uint16\_t total\_blocks;

uint16\_t root\_dir\_index;

uint16\_t data\_block\_index;

uint16\_t total\_data\_blocks;

uint8\_t total\_fat\_blocks;

uint8\_t padding[4079]; // to prevent malloc errors

}\_\_attribute\_\_((\_\_packed\_\_));

struct fat\_block {

uint16\_t entries[2048];

}\_\_attribute\_\_((\_\_packed\_\_));

struct root {

uint8\_t filename[FS\_FILENAME\_LEN];

uint32\_t filesize;

uint16\_t first\_db\_num;

uint8\_t padding[10]; // to prevent malloc issues

}\_\_attribute\_\_((\_\_packed\_\_));

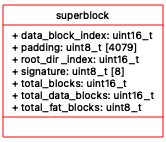
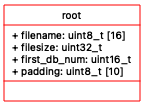
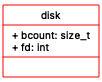
struct fd {

int id;

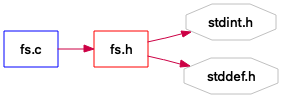
int offset;

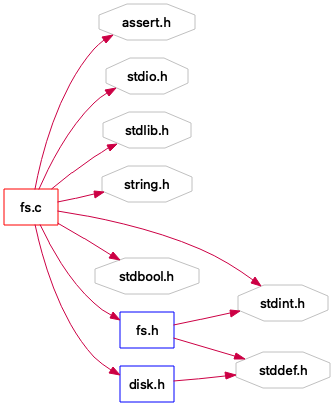
int root\_entry;

}\_\_attribute\_\_((\_\_packed\_\_));

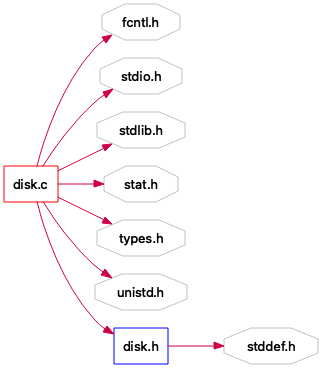
 

### 程序依赖情况如下

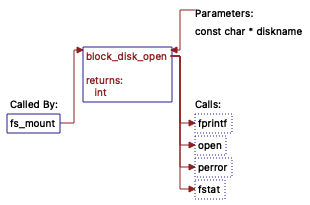


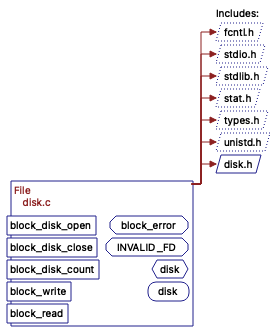


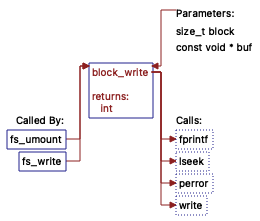
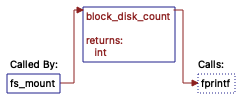


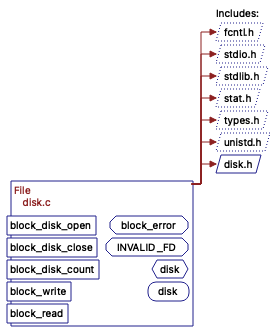
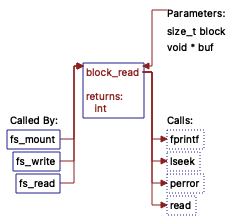


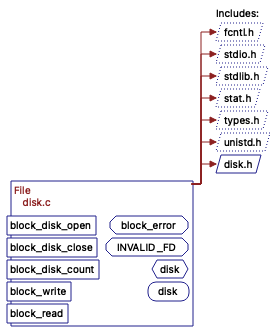
### 函数情况如下



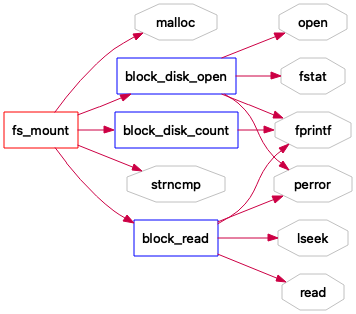


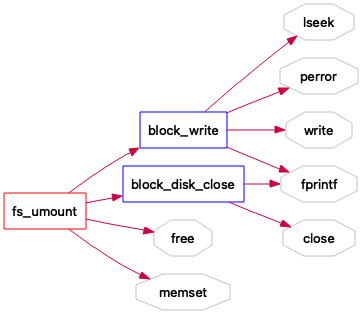


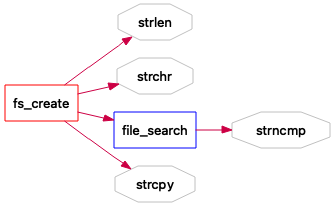


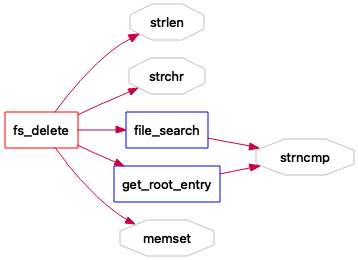


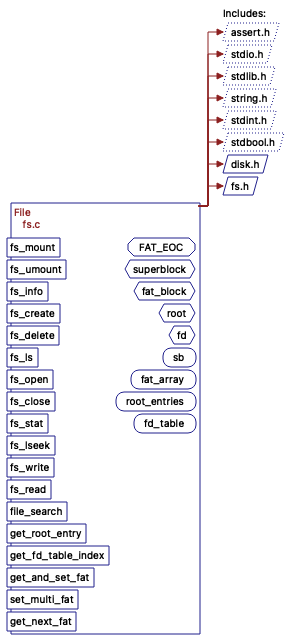
函数声明情况

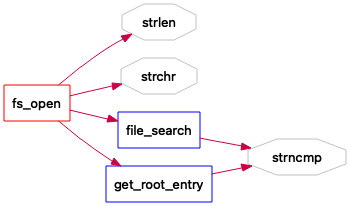


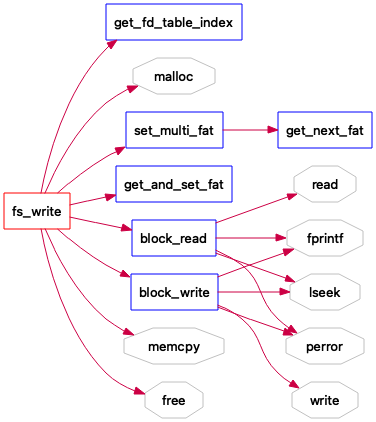


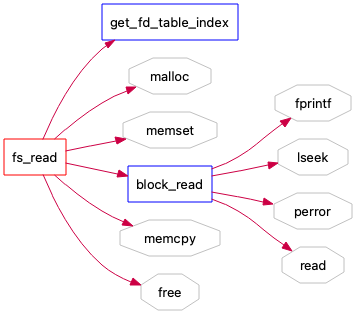






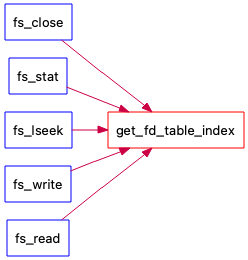












## 程序设计

Disk.c

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include "disk.h"

#define block\_error(fmt, ...) \

fprintf(stderr, "%s: "fmt"\n", \_\_func\_\_, ##\_\_VA\_ARGS\_\_)

/\* Invalid file descriptor \*/

#define INVALID\_FD -1

/\* Disk instance description \*/

struct disk {

/\* File descriptor \*/

int fd;

/\* Block count \*/

size\_t bcount;

};

// 虚拟磁盘

static struct disk disk = { .fd = INVALID\_FD };

int block\_disk\_open(const char \*diskname)

{

int fd;

struct stat st;

if (!diskname) {

block\_error("invalid file diskname");

return -1;

}

if (disk.fd != INVALID\_FD) {

block\_error("disk already open");

return -1;

}

if ((fd = open(diskname, O\_RDWR, 0644)) < 0) {

perror("open");

return -1;

}

if (fstat(fd, &st)) {

perror("fstat");

return -1;

}

// 磁盘大小必须能被整除

if (st.st\_size % BLOCK\_SIZE != 0) {

block\_error("size '%zu' is not multiple of '%d'",

st.st\_size, BLOCK\_SIZE);

return -1;

}

disk.fd = fd;

disk.bcount = st.st\_size / BLOCK\_SIZE;

return 0;

}

// 关闭磁盘

int block\_disk\_close(void)

{

if (disk.fd == INVALID\_FD) {

block\_error("no disk currently open");

return -1;

}

close(disk.fd);

disk.fd = INVALID\_FD;

return 0;

}

int block\_disk\_count(void)

{

if (disk.fd == INVALID\_FD) {

block\_error("no disk currently open");

return -1;

}

return disk.bcount;

}

int block\_write(size\_t block, const void \*buf)

{

// 磁盘还没被打开

if (disk.fd == INVALID\_FD) {

block\_error("no disk currently open");

return -1;

}

// 块数量超过磁盘大小

if (block >= disk.bcount) {

block\_error("block index out of bounds (%zu/%zu)",

block, disk.bcount);

return -1;

}

// 一切正确，位移到相应的块位置

if (lseek(disk.fd, block \* BLOCK\_SIZE, SEEK\_SET) < 0) {

perror("lseek");

return -1;

}

// 写入数据

if (write(disk.fd, buf, BLOCK\_SIZE) < 0) {

perror("write");

return -1;

}

return 0;

}

// 读取数据块

int block\_read(size\_t block, void \*buf)

{

if (disk.fd == INVALID\_FD) {

block\_error("no disk currently open");

return -1;

}

if (block >= disk.bcount) {

block\_error("block index out of bounds (%zu/%zu)",

block, disk.bcount);

return -1;

}

// 位移到相应块

if (lseek(disk.fd, block \* BLOCK\_SIZE, SEEK\_SET) < 0) {

perror("lseek");

return -1;

}

// 读取到buf

if (read(disk.fd, buf, BLOCK\_SIZE) < 0) {

perror("read");

return -1;

}

return 0;

}

Disk.h

#ifndef \_DISK\_H

#define \_DISK\_H

#include <stddef.h>

// 块大小

#define BLOCK\_SIZE 4096

int block\_disk\_open(const char \*diskname);

int block\_disk\_close(void);

int block\_disk\_count(void);

int block\_write(size\_t block, const void \*buf);

int block\_read(size\_t block, void \*buf);

#endif /\* \_DISK\_H \*/

Fs.c

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <stdbool.h>

#include "disk.h"

#include "fs.h"

#define FAT\_EOC 65535

int file\_search(const char\* filename);

int get\_root\_entry(const char\* filename);

int get\_fd\_table\_index(int fd);

size\_t get\_and\_set\_fat(size\_t first\_db\_num);

int set\_multi\_fat (size\_t \*first\_db\_num, size\_t offset\_data\_block);

int get\_next\_fat(int updated\_i, int updated\_j);

// 超级快数据

struct superblock {

uint8\_t signature[8]; // ECS150FS

uint16\_t total\_blocks;

uint16\_t root\_dir\_index;

uint16\_t data\_block\_index;

uint16\_t total\_data\_blocks;

uint8\_t total\_fat\_blocks;

uint8\_t padding[4079]; // 防止溢出

}\_\_attribute\_\_((\_\_packed\_\_));

struct fat\_block {

uint16\_t entries[2048];

}\_\_attribute\_\_((\_\_packed\_\_));

struct root {

uint8\_t filename[FS\_FILENAME\_LEN];

uint32\_t filesize;

uint16\_t first\_db\_num;

uint8\_t padding[10]; // to prevent malloc issues

}\_\_attribute\_\_((\_\_packed\_\_));

struct fd {

int id;

int offset;

int root\_entry;

}\_\_attribute\_\_((\_\_packed\_\_));

static struct superblock\* sb = NULL;

static struct fat\_block\* fat\_array = NULL;

static struct root root\_entries[FS\_FILE\_MAX\_COUNT];// 128 for 1 root block

static struct fd fd\_table[FS\_OPEN\_MAX\_COUNT]; // maximum 32 fd's open at a time

int fs\_mount(const char \*diskname)

{

sb = malloc(sizeof(struct superblock));

if (block\_disk\_open(diskname) == -1) {

return -1;

}

if (block\_read(0, sb) == -1) {

return -1;

}

// 签名

if (strncmp((char \*)sb->signature, "ECS150FS", 8) != 0) {

return -1;

}

if (block\_disk\_count() != sb->total\_blocks) {

return -1;

}

fat\_array = malloc(sb->total\_fat\_blocks \* sizeof(struct fat\_block));

int total\_fat\_counter = (int)sb->total\_fat\_blocks;

size\_t read\_counter = 1;

while (total\_fat\_counter != 0) {

if (block\_read(read\_counter,

fat\_array[read\_counter-1].entries) == -1) {

return -1;

}

read\_counter++;

--total\_fat\_counter;

}

if (fat\_array[0].entries[0] != FAT\_EOC) {

return -1;

}

if (block\_read((size\_t)sb->root\_dir\_index, root\_entries) == -1) {

return -1;

}

for(int i = 0; i < FS\_OPEN\_MAX\_COUNT; ++i) {

fd\_table[i].id = -1; // set all to -1, b/c none has been opened

fd\_table[i].offset = 0; //always init as 0

}

return 0;

}

int fs\_umount(void)

{

if (!sb) {

return -1;

}

for (int i = 0; i < FS\_OPEN\_MAX\_COUNT; ++i) {

if (fd\_table[i].id != -1) {

return -1;

}

}

if (block\_write(0, sb) == -1) {

return -1;

}

for (size\_t i = 0; i < sb->total\_fat\_blocks; i++) {

if(block\_write((i+1), fat\_array[i].entries) == -1) {

return -1;

}

}

if (block\_write(sb->root\_dir\_index, root\_entries) == -1) {

return -1;

}

if (block\_disk\_close() == -1) {

return -1;

}

free(sb);

free(fat\_array);

memset(root\_entries, 0, BLOCK\_SIZE);

memset(fd\_table, 0, sizeof(struct fd)\*FS\_OPEN\_MAX\_COUNT);

sb = NULL;

fat\_array = NULL;

return 0;

}

int fs\_info(void)

{

if (!sb) {

return -1;

}

int fat\_occupied\_count = 0;

for (int i = 0; i < sb->total\_fat\_blocks; ++i) {

for (int j = 0; j < 2048; ++j) { // 2048 entries per FAT block

if (fat\_array[i].entries[j] != 0) {

++fat\_occupied\_count;

}

}

}

int root\_entry\_free\_count = 0;

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (root\_entries[i].filename[0] == '\0') {

++root\_entry\_free\_count;

}

}

printf("FS Info:\n");

printf("total\_blk\_count=%d\n", sb->total\_blocks);

printf("fat\_blk\_count=%d\n", sb->total\_fat\_blocks);

printf("rdir\_blk=%d\n", sb->root\_dir\_index);

printf("data\_blk=%d\n", sb->data\_block\_index);

printf("data\_blk\_count=%d\n", sb->total\_data\_blocks);

printf("fat\_free\_ratio=%d/%d\n",

sb->total\_data\_blocks - fat\_occupied\_count, sb->total\_data\_blocks);

printf("rdir\_free\_ratio=%d/%d\n",

root\_entry\_free\_count, FS\_FILE\_MAX\_COUNT);

return 0;

}

int fs\_create(const char \*filename)

{

if (!sb) {

return -1;

}

if (strlen(filename) >= FS\_FILENAME\_LEN || strlen(filename) == 0) {

return -1;

}

if (!strchr(filename, '\0')) {

return -1;

}

if (file\_search(filename) == 0) {

return -1;

}

int file\_counter = 0; // temporary variable

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (root\_entries[i].filename[0] != '\0') {

++file\_counter;

}

}

if (file\_counter == 128) {

return -1;

}

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (root\_entries[i].filename[0] == '\0') {

strcpy((char \*)root\_entries[i].filename, filename);

root\_entries[i].filesize = 0;

root\_entries[i].first\_db\_num = FAT\_EOC; // fat\_EOC

break;

}

}

return 0;

}

int fs\_delete(const char \*filename)

{

if (!sb) {

return -1;

}

// Check if file name is invalid

if (strlen(filename) >= FS\_FILENAME\_LEN || strlen(filename) == 0) {

return -1;

}

// check if filename is null-terminated

if (!strchr(filename, '\0')) {

return -1;

}

// checks if filename is not found

if (file\_search(filename) != 0) {

return -1;

}

int entry = get\_root\_entry(filename);

size\_t first\_db\_num = root\_entries[entry].first\_db\_num;

size\_t db\_index = first\_db\_num;

db\_index += sb->root\_dir\_index;

++db\_index; // skip over DB #0

size\_t first\_db\_num\_cpy = first\_db\_num; // for the while loop below

int fat\_block\_index = 0;

if (first\_db\_num\_cpy >= 2048) {

while (first\_db\_num\_cpy != 0 && first\_db\_num\_cpy != FAT\_EOC) {

++fat\_block\_index;

first\_db\_num\_cpy /= 2048;

}

}

int root\_index = get\_root\_entry(filename);

int amt\_data\_blocks = root\_entries[root\_index].filesize / BLOCK\_SIZE + 1;

if (amt\_data\_blocks > 1) {

size\_t next\_entry = first\_db\_num;

size\_t cur\_entry = next\_entry;

while (amt\_data\_blocks != 0) {

next\_entry = fat\_array[fat\_block\_index].entries[cur\_entry];

fat\_array[fat\_block\_index].entries[cur\_entry] = 0;

cur\_entry = next\_entry;

amt\_data\_blocks--;

}

}

else {

if (fat\_array[fat\_block\_index].entries[first\_db\_num] == FAT\_EOC) {

fat\_array[fat\_block\_index].entries[first\_db\_num] = 0;

}

}

// freeing the root entry

memset((char \*)root\_entries[entry].filename, 0, 16);

root\_entries[entry].filename[0] = '\0';

root\_entries[entry].first\_db\_num = 0;

root\_entries[entry].filesize = 0;

return 0;

}

int fs\_ls(void)

{

// sb being NULL implies nothing was mounted,

// since sb gets populated in fs\_mount()

if (!sb) {

return -1;

}

printf("FS Ls:\n");

for (size\_t i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (root\_entries[i].filename[0] != '\0') {

printf("file: %s, size: %d, data\_blk: %d\n",

root\_entries[i].filename, root\_entries[i].filesize,

root\_entries[i].first\_db\_num);

}

}

return 0;

}

int fs\_open(const char \*filename) {

if (!sb) {

return -1;

}

// 检查文件名

if (strlen(filename) > FS\_FILENAME\_LEN || strlen(filename) == 0) {

return -1;

}

// 检查文件名末尾

if (!strchr(filename, '\0')) {

return -1;

}

// 查找文件名

if (file\_search(filename) != 0) {

return -1;

}

// 文件未被打开

for(int i = 0; i < FS\_OPEN\_MAX\_COUNT; ++i) {

if (fd\_table[i].id == -1) {

fd\_table[i].id = i;

fd\_table[i].root\_entry = get\_root\_entry(filename);

return fd\_table[i].id;

}

}

return -1;

}

int fs\_close(int fd)

{

if (!sb) {

return -1;

}

if (fd < 0) {

return -1;

}

int fd\_index = get\_fd\_table\_index(fd);

if (fd\_index == -1) {

return -1; // 文件未被打开

}

fd\_table[fd\_index].id = -1;

return 0;

}

int fs\_stat(int fd)

{

if (!sb) {

return -1;

}

if (fd < 0) {

return -1;

}

int fd\_index = get\_fd\_table\_index(fd);

if (fd\_index == -1) {

return -1;

}

return root\_entries[fd\_table[fd\_index].root\_entry].filesize;

}

int fs\_lseek(int fd, size\_t offset)

{

if (!sb) {

return -1;

}

if (fd < 0) {

return -1;

}

// 文件描述符不存在

int fd\_index = get\_fd\_table\_index(fd);

if(fd\_index == -1) {

return -1;

}

// 偏移量超出了文件大小

if(offset > root\_entries[fd\_table[fd\_index].root\_entry].filesize) {

return -1;

}

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

return 0;

}

int fs\_write(int fd, void \*buf, size\_t count)

{

if (!sb) {

return -1;

}

// 文件描述符不存在

int fd\_index = get\_fd\_table\_index(fd);

if (fd\_index == -1) {

return -1;

}

if (count == 0) {

return 0;

}

root\_entries[fd\_table[fd\_index].root\_entry].filesize += (uint32\_t)count;

size\_t amnt\_data\_blocks = (count / BLOCK\_SIZE) + 1; // due to truncation

size\_t buf\_offset = 0;

size\_t multi\_count = count;

void \*bounce\_buf = malloc(BLOCK\_SIZE);

size\_t first\_db\_num =

root\_entries[fd\_table[fd\_index].root\_entry].first\_db\_num;

size\_t fd\_offset = (size\_t)fd\_table[fd\_index].offset;

uint32\_t filesize = root\_entries[fd\_table[fd\_index].root\_entry].filesize;

size\_t first\_db\_num\_cpy = first\_db\_num; // for the while loop below

int fat\_block\_index = 0;

if (first\_db\_num\_cpy >= 2048 && first\_db\_num\_cpy != FAT\_EOC) {

while (first\_db\_num\_cpy != 0) {

++fat\_block\_index;

first\_db\_num\_cpy /= 2048;

}

}

size\_t block\_offset = fd\_offset/BLOCK\_SIZE;

size\_t byte\_offset = fd\_offset - (block\_offset\*BLOCK\_SIZE);

size\_t offset\_data\_block = amnt\_data\_blocks - block\_offset;

size\_t offset\_db\_index = block\_offset;

// array that holds where fat entries are.

uint16\_t fat\_location[2048 \* sb->total\_fat\_blocks];

size\_t free\_fat = 0;

if (amnt\_data\_blocks > 1) {

if (set\_multi\_fat(&first\_db\_num, offset\_data\_block) == -1) {

return 0; // no more free fat entries to write to

}

root\_entries[fd\_table[fd\_index].root\_entry].first\_db\_num

= (uint16\_t)first\_db\_num;

fat\_location[0] = (uint16\_t) first\_db\_num;

}

else {

free\_fat = get\_and\_set\_fat(first\_db\_num);

fat\_location[0] = (uint16\_t) free\_fat;

}

for(int i = 1; i < amnt\_data\_blocks; i++){

fat\_location[i] = fat\_array[fat\_block\_index].entries[fat\_location[i-1]];

}

if (amnt\_data\_blocks == 1 && first\_db\_num == FAT\_EOC) {

if (free\_fat == 0) {

return 0; // FAT满

}

size\_t db\_index = free\_fat + sb->root\_dir\_index;

++db\_index; // skip over DB #0

// 更新数据块

root\_entries[fd\_table[fd\_index].root\_entry].first\_db\_num

= (uint16\_t) free\_fat;

block\_write(db\_index, buf);

}

else if (count + fd\_offset <= filesize) {

while (offset\_data\_block != 0) {

if(count <= BLOCK\_SIZE - byte\_offset){

if (block\_read(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1,

bounce\_buf) == -1) {

return -1;

}

memcpy(bounce\_buf + byte\_offset, buf,

BLOCK\_SIZE - byte\_offset);

block\_write(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1, bounce\_buf);

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

else {

if(byte\_offset > 0) {

if (block\_read(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1,

bounce\_buf) == -1) {

return -1;

}

memcpy(bounce\_buf + byte\_offset, buf,

BLOCK\_SIZE - byte\_offset);

block\_write(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1, bounce\_buf);

buf\_offset = buf\_offset + (BLOCK\_SIZE - byte\_offset);

multi\_count = multi\_count - (BLOCK\_SIZE - byte\_offset);

offset\_db\_index++;

byte\_offset = 0; //we'll never use this again

// if multi\_count 0, we just return.

if (multi\_count == 0) {

fd\_table[fd\_index].offset += multi\_count;

free(bounce\_buf);

return (int)multi\_count;

}

}

else if (multi\_count > BLOCK\_SIZE) {

memcpy(bounce\_buf, buf+buf\_offset, BLOCK\_SIZE);

block\_write(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1, bounce\_buf);

buf\_offset = buf\_offset + BLOCK\_SIZE;

multi\_count = multi\_count - BLOCK\_SIZE;

offset\_db\_index++;

}

else {

if (block\_read(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1,

bounce\_buf) == -1) {

return -1;

}

memcpy(bounce\_buf, buf+buf\_offset, multi\_count);

block\_write(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1, bounce\_buf);

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int) count;

}

}

offset\_data\_block--;

}

}

fd\_table[fd\_index].offset += count;

return (int)count;

}

int fs\_read(int fd, void \*buf, size\_t count)

{

if (!sb) {

return -1;

}

// invalid fd check

if (fd < 0) {

return -1;

}

// if the fd index doesn't exist, return -1

int fd\_index = get\_fd\_table\_index(fd);

if (fd\_index == -1) {

return -1;

}

// array that holds where fat blocks are.

uint16\_t fat\_location[2048];

size\_t fd\_offset = (size\_t)fd\_table[fd\_index].offset;

uint32\_t filesize = root\_entries[fd\_table[fd\_index].root\_entry].filesize;

size\_t amnt\_data\_blocks = (filesize/BLOCK\_SIZE) + 1;

void \*bounce\_buf = malloc(BLOCK\_SIZE); //used to hold a temp data block.

memset(bounce\_buf, 0, BLOCK\_SIZE);

uint32\_t bytes\_in\_file = filesize;

size\_t db\_index

= root\_entries[fd\_table[fd\_index].root\_entry].first\_db\_num;

++db\_index; // because db counts up from 0

db\_index = db\_index + sb->root\_dir\_index;

size\_t first\_db\_num

= root\_entries[fd\_table[fd\_index].root\_entry].first\_db\_num;

size\_t first\_db\_num\_cpy = first\_db\_num; // for the while loop below

int fat\_block\_index = 0; //get which FAT block DB is located in

if (first\_db\_num\_cpy >= 2048) {

while (first\_db\_num\_cpy != 0) {

++fat\_block\_index;

first\_db\_num\_cpy /= 2048;

}

}

fat\_location[0] = (uint16\_t)first\_db\_num;

for(int i = 1; i < amnt\_data\_blocks; i++){

// we get the index of what the current one is "pointing" to

fat\_location[i] = fat\_array[fat\_block\_index].entries[fat\_location[i-1]];

}

size\_t buf\_offset = 0;

size\_t multi\_count = count;

if (filesize == fd\_offset) {

return 0;

}

size\_t block\_offset = fd\_offset/BLOCK\_SIZE;

size\_t byte\_offset = fd\_offset - (block\_offset\*BLOCK\_SIZE);

size\_t offset\_data\_block = amnt\_data\_blocks - block\_offset;

size\_t offset\_db\_index = block\_offset;

if (fd\_offset != 0) {

bytes\_in\_file = bytes\_in\_file - (uint32\_t)fd\_offset;

while (offset\_data\_block != 0) {

if (buf\_offset == count) {

break;

}

if (block\_read(fat\_location[offset\_db\_index]

+ sb->root\_dir\_index + 1,

bounce\_buf) == -1) {

return -1;

}

if (count <= BLOCK\_SIZE - byte\_offset) {

if (bytes\_in\_file > count) {

memcpy(buf + buf\_offset,

bounce\_buf + byte\_offset,multi\_count);

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

else {

memcpy(buf, bounce\_buf + byte\_offset, bytes\_in\_file);

fd\_table[fd\_index].offset += bytes\_in\_file;

free(bounce\_buf);

return (int)bytes\_in\_file;

}

}

else {

if (byte\_offset > 0) { //should only run this fxn at most ONCE.

memcpy(buf, bounce\_buf + byte\_offset,

(BLOCK\_SIZE- byte\_offset));

buf\_offset = buf\_offset + (BLOCK\_SIZE - byte\_offset);

multi\_count = multi\_count - (BLOCK\_SIZE - byte\_offset);

offset\_db\_index++;

byte\_offset = 0;

if (multi\_count == 0) {

fd\_table[fd\_index].offset += multi\_count;

free(bounce\_buf);

return (int)multi\_count;

}

}

// so if multi\_count is still greater than BLOCK\_SIZE,

// we just memcpy the whole block into buf.

else if(multi\_count > BLOCK\_SIZE) {

memcpy(buf + buf\_offset, bounce\_buf, BLOCK\_SIZE);

buf\_offset = buf\_offset + BLOCK\_SIZE;

multi\_count = multi\_count - BLOCK\_SIZE;

offset\_db\_index++;

}

// multi\_count less than BLOCK\_SIZE

// we read what's left of the block.

else {

memcpy(buf + buf\_offset, bounce\_buf, multi\_count);

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

}

offset\_data\_block--;

} // end of while loop

if (fd\_offset + count > filesize) {

fd\_table[fd\_index].offset += bytes\_in\_file;

free(bounce\_buf);

return (int)bytes\_in\_file;

}

else {

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

} // end of if

int iterator = 0; //used for iterating thru the fat\_location

while (amnt\_data\_blocks != 0) {

if (buf\_offset == count) {

break;

}

if (block\_read((fat\_location[iterator] + sb->root\_dir\_index + 1),

bounce\_buf) == -1) {

free(bounce\_buf);

return -1;

}

// if count < or = BLOCK\_SIZE, we just read in everything to buf.

if (count <= BLOCK\_SIZE) {

if (count > bytes\_in\_file) {

// here is if count > file size.

memcpy(buf, bounce\_buf, bytes\_in\_file);

fd\_table[fd\_index].offset += bytes\_in\_file;

free(bounce\_buf);

return (int) bytes\_in\_file;

}

else {

memcpy(buf, bounce\_buf, count);

//we modify the offset before returning.

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

}

else {

if (multi\_count > BLOCK\_SIZE) {

if (multi\_count > bytes\_in\_file

&& bytes\_in\_file < BLOCK\_SIZE) {

// here, we've read the maximum amount of blocks

// we then read what we can of the last block,

// and then finally return.

memcpy(buf+buf\_offset, bounce\_buf, bytes\_in\_file);

buf\_offset = buf\_offset + bytes\_in\_file;

fd\_table[fd\_index].offset += filesize;

free(bounce\_buf);

return (int)filesize;

}

else {

memcpy(buf+buf\_offset, bounce\_buf, BLOCK\_SIZE);

buf\_offset = buf\_offset + BLOCK\_SIZE;

multi\_count = multi\_count - BLOCK\_SIZE;

bytes\_in\_file = bytes\_in\_file - BLOCK\_SIZE;

db\_index++;

}

}

else {

// There are basically two scenarios here.

// if multi count is the limiter,

// or if filesize is the limiter.

if (multi\_count < bytes\_in\_file) {

memcpy(buf+buf\_offset, bounce\_buf, multi\_count);

buf\_offset = buf\_offset + multi\_count;

db\_index++;

}

else if (bytes\_in\_file <= multi\_count) {

memcpy(buf+buf\_offset, bounce\_buf, bytes\_in\_file);

fd\_table[fd\_index].offset += filesize;

free(bounce\_buf);

return (int)filesize;

}

}

}

--amnt\_data\_blocks;

iterator++;

} // end of while loop

// we modify the offset before returning.

fd\_table[fd\_index].offset += count;

free(bounce\_buf);

return (int)count;

}

/\* HELPER FUNCTIONS \*/

// Find a file named filename that exists inside the root entries.

// Return: -1 if filename was not found in the root entries.

// Otherwise return 0 to indicate file was found.

int file\_search(const char\* filename) {

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (strncmp( (char\*)root\_entries[i].filename,

filename, FS\_FILENAME\_LEN ) == 0) {

return 0; // found a match

}

}

return -1; // fail state: could not find file

}

// Find a file named @filename that exists inside the root entries.

// Return: -1 if @filename was not found in the root entries.

// Otherwise return the root entry index of where @filename

// was located in.

int get\_root\_entry(const char\* filename) {

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

if (strncmp( (char\*)root\_entries[i].filename,

filename, FS\_FILENAME\_LEN ) == 0) {

return i; // found a match

}

}

return -1; // fail state: could not find file

}

// Find a file descriptor named @fd that exists inside

// the fd table array.

// Return: -1 if @fd was not found in the fd table.

// Otherwise return the index of where @fd was

// found in the fd table.

int get\_fd\_table\_index(int fd) {

for (int i = 0; i < FS\_FILE\_MAX\_COUNT; ++i) {

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

return i; // found index

}

}

return -1; // fail state: could not find opened fd

}

// sets the FAT entry for a file that is only 1 data block big.

// returns 0 if it reaches the complete end of the double for loops,

// meaning no more free fat entries available.

size\_t get\_and\_set\_fat(size\_t first\_db\_num) {

for (int i = 0; i < sb->total\_fat\_blocks; ++i) {

for (int j = 0; j < 2048; ++j) { // 2048 entries per FAT block

if (first\_db\_num != FAT\_EOC

&& fat\_array[i].entries[first\_db\_num] == FAT\_EOC) {

return first\_db\_num;

}

if (fat\_array[i].entries[j] == first\_db\_num

&& first\_db\_num != FAT\_EOC) {

return first\_db\_num;

}

if (fat\_array[i].entries[j] == 0) {

// assign the file less than 4096 bytes

// to a singular, proper FAT entry, and set that

// to 0xFFFF.

fat\_array[i].entries[j] = FAT\_EOC;

return (size\_t)j;

}

} // end of j loop

} // end of i loop

return 0; // no free fat\_entries available

// => no free data blocks available.

}

// if the file has n data blocks associated,

// we use this function to assign n FAT entries

int set\_multi\_fat (size\_t \*first\_db\_num, size\_t offset\_data\_block) {

int free\_entry\_check = 0;

for (int i = 0; i < sb->total\_fat\_blocks; ++i) {

if (offset\_data\_block == 0) {

break;

}

for (int j = 0; j < 2048; ++j) {

if (fat\_array[i].entries[j] == 0) {

// case where we have contiguous free FAT entries

if (fat\_array[i].entries[j + 1] == 0) {

fat\_array[i].entries[j] = (uint16\_t) (j+1);

// means first\_db\_num hasn't been set yet

if (\*first\_db\_num == FAT\_EOC) {

// only enters here once

\*first\_db\_num = (size\_t)j;

}

offset\_data\_block--;

}

else {

// non-contiguous case

// need some helper function to iterate from

// where we are now until it finds the next

// free fat entry. Then store this in free\_entry\_check.

free\_entry\_check = get\_next\_fat(i, j+1);

if (free\_entry\_check == -1) {

return -1; // no more free entries

}

fat\_array[i].entries[j] = (uint16\_t)free\_entry\_check;

// means first\_db\_num hasn't been set yet

if (\*first\_db\_num == FAT\_EOC) {

// only enters here once

\*first\_db\_num = (size\_t)j;

}

// update j so we don't iterate through unnecessary

// entries (any entry before get\_next\_fat() )

j = fat\_array[i].entries[j] - 1;

--offset\_data\_block;

}

}

// the last entry should be set to 0xFFFF

if (offset\_data\_block == 0) {

fat\_array[i].entries[j] = FAT\_EOC;

break;

}

} // j loop end

} // i loop end

return 0; // success condition

}

// finds the next free fat entry. The first

// one the function finds, it returns.

int get\_next\_fat(int updated\_i, int updated\_j) {

for (int i = updated\_i; i < sb->total\_fat\_blocks; ++i) {

for (int j = updated\_j; j < 2048; ++j) {

if (fat\_array[i].entries[j] == 0) {

return (uint16\_t)j;

}

}

}

return -1; // no free fat\_entries available

// => no free data blocks available.

}

Fs.h

#ifndef \_FS\_H

#define \_FS\_H

#include <stdint.h>

#include <stddef.h>

/\*\* Maximum filename length (including the NULL character) \*/

#define FS\_FILENAME\_LEN 16

/\*\* Maximum number of files in the root directory \*/

#define FS\_FILE\_MAX\_COUNT 128

/\*\* Maximum number of open files \*/

#define FS\_OPEN\_MAX\_COUNT 32

int fs\_mount(const char \*diskname);

int fs\_umount(void);

int fs\_info(void);

int fs\_create(const char \*filename);

int fs\_delete(const char \*filename);

int fs\_ls(void);

int fs\_open(const char \*filename);

int fs\_close(int fd);

int fs\_stat(int fd);

int fs\_lseek(int fd, size\_t offset);

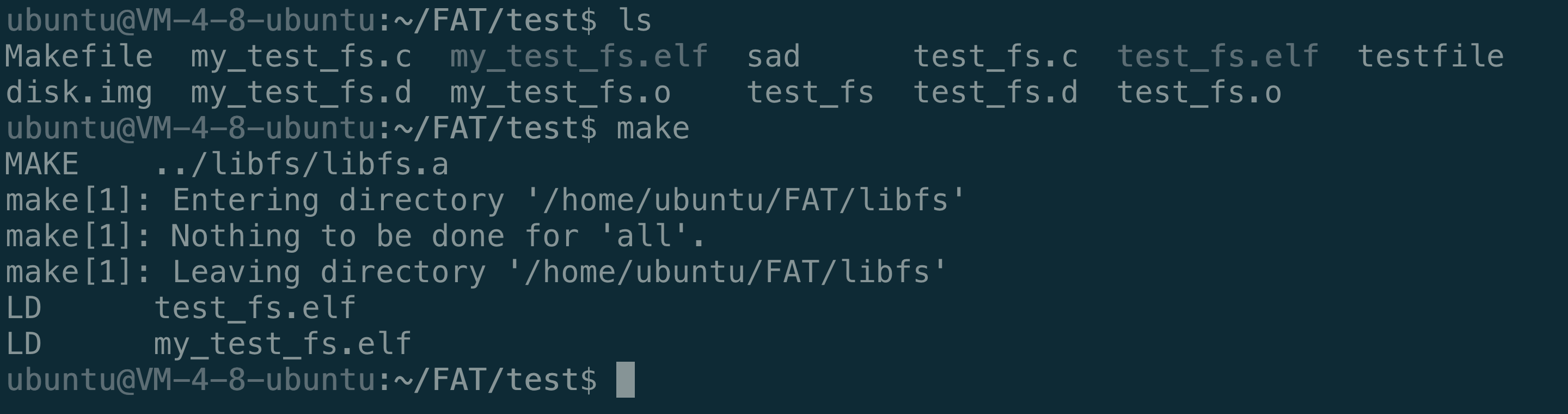
int fs\_write(int fd, void \*buf, size\_t count);

int fs\_read(int fd, void \*buf, size\_t count);

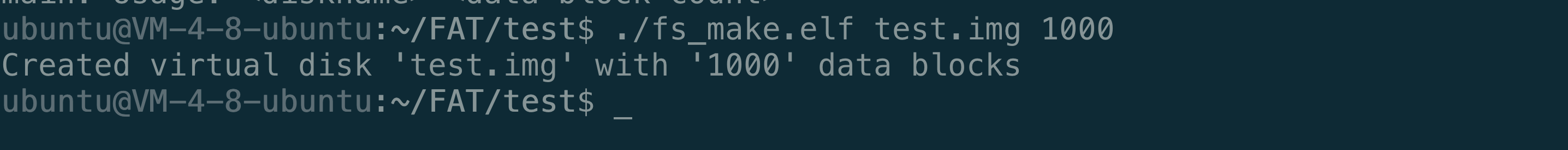
#endif /\* \_FS\_H \*/

## 运行截图

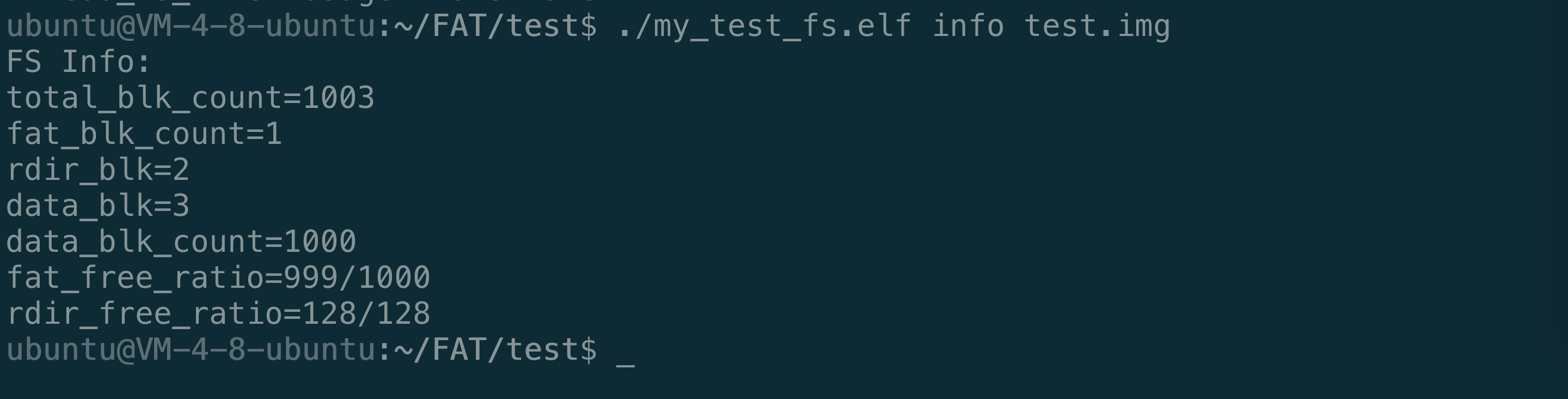
### make编译



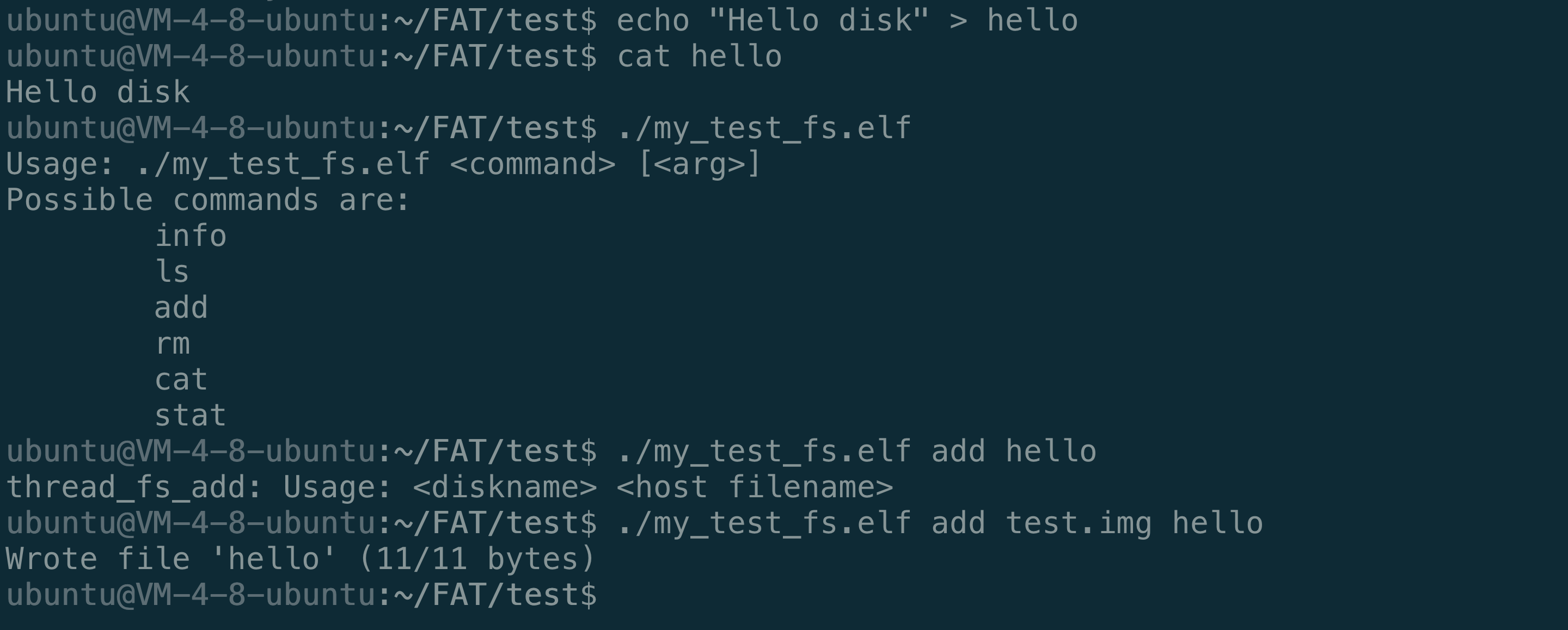
### 创建img镜像



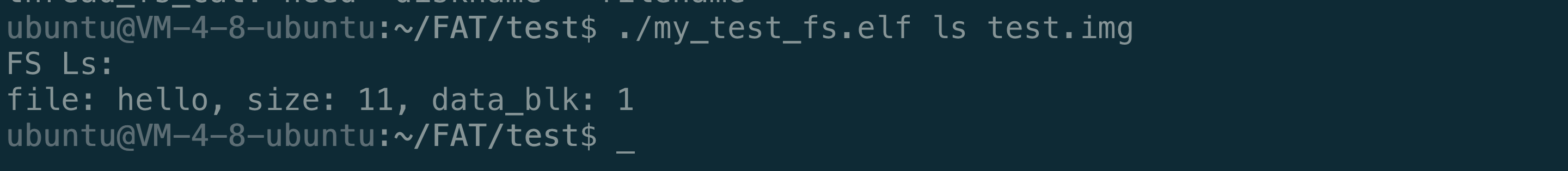
### 查看镜像信息



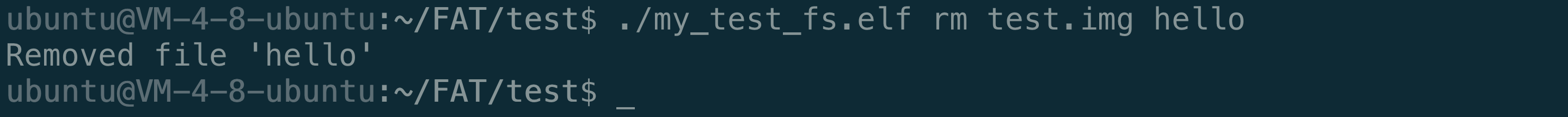
### 创建测试文件并写入硬盘



### 查看保存的文件



### 删除文件



### 检验

