

浙江大学

本科实验报告

课程名称：操作系统

姓名：夏尤楷

学院：计算机科学与技术学院

系：计算机科学与技术系

专业：计算机科学与技术

学号：3210104331

指导教师：夏莹杰

2024 年 1 月 14 日

浙江大学操作系统实验报告

实验名称: VFS & FAT32 文件系统

电子邮件地址: 459510812@qq.com 手机: 15058004449

实验地点: 玉泉曹光彪西楼 503 实验日期: 2024 年 1 月 14 日

一、实验目的和要求

1. 为用户态的 Shell 提供 read 和 write syscall 的实现。
2. 实现 FAT32 文件系统的基本功能，并对其中的文件进行读写。

二、实验过程

(一) Shell: 与内核进行交互

1. 准备工作

同步 os23fall-stu 中的 user 文件夹，替换原有的用户态程序为 nish。为了能够正确启动 QEMU，需要下载磁盘镜像并放置在项目目录下。将文件按如下方式放置。

```
lab7
├── Makefile
├── disk.img
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       └── sbi.h
├── fs
│   ├── Makefile
│   ├── fat32.c
│   ├── fs.S
│   ├── mbr.c
│   ├── vfs.c
│   └── virtio.c
├── include
│   ├── fat32.h
│   ├── fs.h
│   ├── mbr.h
│   ├── string.h
│   └── debug.h
```

```

├── virtio.h
├── lib
│   └── string.c
├── user
│   ├── Makefile
│   ├── forktest.c
│   ├── link.lds
│   ├── printf.c
│   ├── ramdisk.S
│   ├── shell.c
│   ├── start.S
│   ├── stddef.h
│   ├── stdio.h
│   ├── string.h
│   ├── syscall.h
│   ├── unistd.c
│   └── unistd.h

```

此外，向 `include/types.h` 中补充一些类型别名：

```

1. typedef unsigned long uint64_t;
2. typedef long int64_t;
3. typedef unsigned int uint32_t;
4. typedef int int32_t;
5. typedef unsigned short uint16_t;
6. typedef short int16_t;
7. typedef uint64_t* pagetable_t;
8. typedef char int8_t;
9. typedef unsigned char uint8_t;
10. typedef uint64_t size_t;

```

将一些源文件中的定义和声明打包成头文件（比如 `vm.h`、`clock.h`），将一些功能（比如虚拟地址转换为物理地址）打包成函数。

2. 处理 `stdout` 的写入

在 `syscall.h` 中加入系统调用声明：

```

1. // arch/riscv/include/syscall.h
2. int64_t sys_write(unsigned int fd, const char* buf, uint64_t count);

```

并在 `trap_handler` 中添加相应入口：

```

1. void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs* regs) {
2.     ...
3.     else if (scause == 8) {
4.         uint64_t ret;
5.         uint64_t syscall_id = regs->x[17];
6.         ...
7.         else if (syscall_id == SYS_WRITE) {
8.             int fd = (int)regs->x[10];
9.             char* buffer = (char*)regs->x[11];
10.            uint64_t siz = (uint64_t)regs->x[12];
11.
12.            ret = (uint64_t)sys_write(fd, buffer, siz);

```

```

13.     }
14.     ...
15.     regs->x[10] = ret;
16.     regs->sepc += 4;
17. }
18. ...
19. }

```

在 `proc.h` 的 `task_struct` 结构体定义中添加对文件表的声明：

```

1. // arch/riscv/include/proc.h
2. struct task_struct {
3.     ...
4.     struct file *files;
5.     ...
6. };

```

在创建进程时为进程初始化文件，当初始化进程时，先完成打开的文件的列表的初始化，这里我们的方式是直接分配一个页（在 `file_init` 中实现），并用 `files` 指向这个页：

```

1. // arch/riscv/kernel/proc.c
2. void task_init() {
3.     ...
4.     // Initialize the stdin, stdout, and stderr.
5.     task[1]->files = file_init();
6.     printk("[S] proc_init done!\n");
7.     ...
8. }

```

其中，`file_init` 中包含对 `stdin`、`stdout`、`stderr` 的初始化。补全对 `stdout` 的初始化：

```

1. // stdout
2. ret[1].opened = 1;
3. ret[1].perms = FILE_WRITABLE;
4. ret[1].cfo = 0;
5. ret[1].lseek = NULL;
6. ret[1].write = stdout_write;
7. ret[1].read = NULL;
8. memcpy(ret[1].path, "stdout", 7);

```

其中，赋给 `ret[1].write` 函数指针值的函数的定义 `stdout_write` 已经给出。

接着实现 `sys_write syscall`，来间接调用我们赋值的 `stdout` 对应的函数指针：

```

1. // arch/riscv/kernel/syscall.c
2. int64_t sys_write(unsigned int fd, const char* buf, size_t count)
3. {
4.     int64_t ret;
5.     struct file* target_file = &(current->files[fd]);
6.     if (target_file->opened) {
7.         /* todo: indirect call */
8.         if (target_file->perms & FILE_WRITABLE)
9.             ret = target_file->write(target_file, buf, count);
10.    } else {
11.        printk("file not open\n");
12.        ret = ERROR_FILE_NOT_OPEN;
13.    }
14.    return ret;
15.}

```

3. 处理 stderr 的写入

仿照 stdout 的输出过程, 完成 stderr 的写入, 具体代码实现如下:

```

1. // fs/vfs.c
2. struct file* file_init() {
3.     ...
4.     // stderr
5.     ret[2].opened = 1;
6.     ret[2].perms = FILE_WRITABLE;
7.     ret[2].cfo = 0;
8.     ret[2].lseek = NULL;
9.     ret[2].write = stderr_write;
10.    ret[2].read = NULL;
11.    memcpy(ret[2].path, "stderr", 7);
12.
13.    return ret;
14.}
15.
16.int64_t stderr_write(struct file* file, const void* buf, uint64_t len) {
17.    /* todo */
18.    char to_print[len + 1];
19.    for (int i = 0; i < len; i++) {
20.        to_print[i] = ((const char*)buf)[i];
21.    }
22.    to_print[len] = 0;
23.    return printk(buf);
24.}

```

4. 处理 stdin 的读取

此时 nish 已经打印出命令行等待输入命令以进行交互了，但是还需要读入从终端输入的命令才能够与人进行交互，所以我们要实现 `stdin` 以获取键盘键入的内容。

代码框架中已经实现了一个在内核态用于向终端读取一个字符的函数 `uart_getchar`，调用这个函数来实现 `stdin_read`。同时在 `file_init` 中完成对 `stdin` 的设置，代码如下：

```
1. // fs/vfs.c
2. struct file* file_init() {
3.     struct file *ret = (struct file*)alloc_page();
4.
5.     // stdin
6.     ret[0].opened = 1;
7.     ret[0].perms = FILE_READABLE;
8.     ret[0].cfo = 0;
9.     ret[0].lseek = NULL;
10.    ret[0].write = NULL;
11.    ret[0].read = stdin_read;
12.    memcpy(ret[0].path, "stdin", 6);
13.    ...
14.    return ret;
15.}
16.
17.int64_t stdin_read(struct file* file, void* buf, uint64_t len) {
18.    /* todo: use uart_getchar() to get <len> chars */
19.    for (unsigned int i = 0; i < len; ++i) {
20.        ((char *)buf)[i] = uart_getchar();
21.    }
22.    return len;
23.}
```

接着参考 `syscall_write` 的实现，来实现 `syscall_read`，并在 `trap_handler` 中为 `syscall_read` 设置相应入口。`syscall_read` 代码如下：

```
1. // arch/riscv/kernel/syscall.c
2.
3. int64_t sys_read(unsigned int fd, char* buf, uint64_t count) {
4.     int64_t ret;
5.     struct file* target_file = &(current->files[fd]);
6.     if (target_file->opened) {
7.         /* todo: indirect call */
8.         if (target_file->perms & FILE_READABLE)
9.             ret = target_file->read(target_file, buf, count);
```

```

10.     } else {
11.         printk("file not open\n");
12.         ret = ERROR_FILE_NOT_OPEN;
13.     }
14.     return ret;
15. }

```

至此，就可以在 nish 中使用 echo 命令了。

（二）FAT32: 持久存储

1. 准备工作

在 setup_vm_final 中创建虚拟内存映射时,添加映射 VritIO 外设部分的映射:

```

1. // arch/riscv/kernel/vm.c
2. create_mapping(swapper_pg_dir, io_to_virt(VIRTIO_START), VIRTIO_START,
    VIRTIO_SIZE * VIRTIO_COUNT, PTE_W | PTE_R | PTE_V);

```

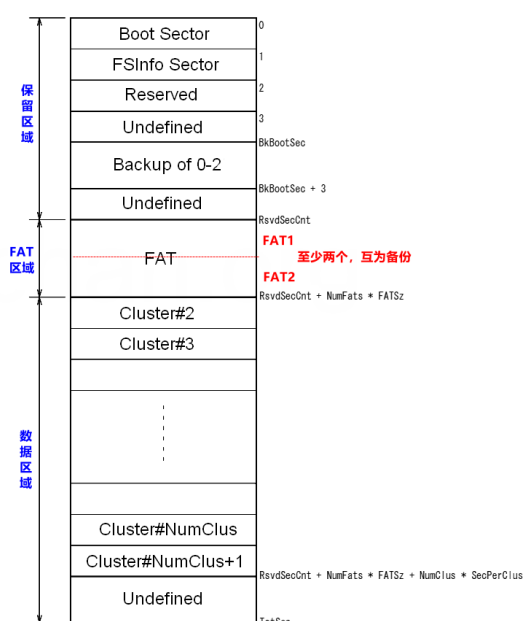
将对 VirtIO 和 MBR 进行初始化的逻辑添加在初始化第一个进程的 task_init 中,代码如下:

```

1. void task_init() {
2.     ...
3.     printk("[S] proc_init done!\n");
4.
5.     virtio_dev_init();
6.     mbr_init();
7.
8.     for (int i = 2; i < NR_TASKS; ++i)
9.         task[i] = NULL;
10.
11.     return;
12. }

```

FAT32 Volume



初始化 FAT32 分区。在 FAT32 分区的第一个扇区中存储了关于这个分区的元数据, 首先需要将该扇区的内容读到一个 fat32_bpb 数据结构中进行解析。fat32_volume 是用来存储我们实验中需要用到的元数据的, 需要根据 fat32_bpb 中的数据来进行计算并初始化。

计算和初始化的完成参考左图。左图显

示 FAT32 的分区依次分为保留区域、FAT 区域和数据区域。观察 fat32_bpb 和 fat32_volume 各成员变量的名称，推测其含义，容易得到初始化的函数实现如下：

```
1. void fat32_init(uint64_t lba, uint64_t size) {
2.     virtio_blk_read_sector(lba, (void*)&fat32_header);
3.     fat32_volume.first_fat_sec = lba + fat32_header.rsvd_sec_cnt;
4.     fat32_volume.sec_per_cluster = fat32_header.sec_per_clus;
5.     fat32_volume.first_data_sec = fat32_volume.first_fat_sec +
        fat32_header.fat_sz32 * fat32_header.num_fats;
6.     fat32_volume.fat_sz = fat32_header.fat_sz32;
7.
8.     virtio_blk_read_sector(fat32_volume.first_data_sec, fat32_buf);
        // Get the root directory
9.     struct fat32_dir_entry *dir_entry = (struct fat32_dir_entry *)fat32_buf;
10. }
```

2. 实现 FAT32 文件的打开与关闭

在读、写文件之前，首先需要打开对应的文件，这要实现 `openat` `syscall`。往 `syscall.c` 和 `syscall.h` 中添加相应定义和声明，并在 `trap_handler` 中设置其入口。`sys_openat` 定义如下：

```
1. // arch/riscv/syscall.c
2.
3. int64_t sys_openat(int dfd, const char* filename, int flags) {
4.     int fd = -1;
5.
6.     // Find an available file descriptor first
7.     for (int i = 0; i < PGSIZE / sizeof(struct file); i++) {
8.         if (!current->files[i].opened) {
9.             fd = i;
10.            break;
11.        }
12.    }
13.
14.    // Do actual open
15.    file_open(&(current->files[fd]), filename, flags);
16.
17.    return fd;
18. }
```

完成 `fat32_open_file`，实现最简单的判别文件系统的方式。文件前缀为“/fat32/”的即是本次 FAT32 文件系统中的文件，在使用命令

“cat /fat32/\$FILENAME” 读取文件时，file_open 会根据前缀决定是否调用 fat32_open_file 函数。因为我们的文件一定在根目录（在第 2 簇中）下，也即 “/fat32/” 下，无需实现与目录遍历相关的逻辑。目录由一条一条的项（entry）构成，遍历根目录簇中的每一项，项内的文件名和 path 中的文件名比对，找到对应文件名的文件，返回文件位置及其目录所处位置的信息。注意：（1）比较文件名时要把文件名统一转换为大或小写，因为这里比对文件名时不区分大小写；（2）path 中文件名字符串结尾以 “\0” 结束，而 entry→name[8] 字符串末尾未被字符填满的部分全部用空格补足，要将两字符串的结尾填充改为相同才方便比较（这里改为全为空格）。fat32_open_file 实现如下：

```
1. // arch/riscv/syscall.c
2. struct fat32_file fat32_open_file(const char *path) {
3.     struct fat32_file file;
4.     /* todo: open the file according to path */
5.     char filename[9];
6.     memset(filename, ' ', 9);
7.     filename[8] = '\0';
8.
9.     if (strlen(path) - 7 > 8)
10.         memcpy(filename, path + 7, 8);
11.     else
12.         memcpy(filename, path + 7, strlen(path) - 7);
13.     to_upper_case(filename);
14.
15.     uint64_t sector = fat32_volume.first_data_sec;
16.
17.     virtio_blk_read_sector(sector, fat32_buf);
18.     struct fat32_dir_entry *entry = (struct fat32_dir_entry *)fat32_buf;
19.
20.     for (int entry_index = 0; entry_index < fat32_volume.sec_per_cluster
21.         * FAT32_ENTRY_PER_SECTOR; ++entry_index) {
22.         char name[8];
23.         memcpy(name, entry->name, 8);
24.         for (int k = 0; k < 9; ++k) //to upper case
25.             if (name[k] <= 'z' && name[k] >= 'a')
26.                 name[k] += 'A' - 'a';
27.         if (memcmp(filename, name, 8) == 0) {
28.             file.cluster = ((uint32_t)entry->starthi << 16) | entry->startlow;
```

```

28.         file.dir.index = entry_index;
29.         file.dir.cluster = 2;
30.         return file;
31.     }
32.     ++entry;
33. }
34. printk("[S] file not found!\n");
35.
36. return file;
37.}

```

在打开文件后自然是进行文件的读取操作，需要先实现 `lseek` syscall（并在 `trap_handler` 中添加相应入口）。FAT32 文件使用 `fat32_lseek(struct file* file, int64_t offset, uint64_t whence)` 以更新指向的文件中的位置，其中当 `whence` 为 `SEEK_SET` 时，更新指向位置为从文件起始地址偏移 `offset` 字节的位置；当 `whence` 为 `SEEK_CUR` 时，更新指向位置为从当前指向的文件中的位置偏移 `offset` 字节的位置；当 `whence` 为 `SEEK_END` 时，更新指向位置为从文件结束地址偏移 `offset` 字节的位置，实现如下：

```

1. // arch/riscv/kernel/syscall.c
2.
3. int64_t sys_lseek(int fd, int64_t offset, int whence) {
4.     int64_t ret;
5.     struct file* target_file = &(current->files[fd]);
6.     if (target_file->opened) {
7.         /* todo: indirect call */
8.         if (target_file->perms & FILE_READABLE)
9.             ret = target_file->lseek(target_file, offset, whence);
10.    } else {
11.        printk("file not open\n");
12.        ret = ERROR_FILE_NOT_OPEN;
13.    }
14.    return ret;
15.}
16.
17. // fs/fat32.c
18.
19. int64_t fat32_lseek(struct file* file, int64_t offset, uint64_t whence) {
20.     if (whence == SEEK_SET) {
21.         file->cfo = offset;
22.     } else if (whence == SEEK_CUR) {
23.         file->cfo = file->cfo + offset;
24.     } else if (whence == SEEK_END) {
25.         /* Calculate file length */
26.         file->cfo = get_file_size(file) + offset;
27.     } else {

```

```

28.     printk("fat32_lseek: whence not implemented\n");
29.     while (1);
30. }
31. return file->cfo;
32. }

```

其中 `get_file_size` 是通过文件的目录项获得文件大小的自己实现的函数：

```

1. uint32_t get_file_size(struct file* file) {
2.     uint64_t sector = cluster_to_sector(file->fat32_file.dir.cluster)
        + file->fat32_file.dir.index / FAT32_ENTRY_PER_SECTOR;
3.     virtio_blk_read_sector(sector, fat32_table_buf);
4.     uint32_t index = file->fat32_file.dir.index % FAT32_ENTRY_PER_SECTOR;
5.     return ((struct fat32_dir_entry *)fat32_table_buf)[index].size;
6. }

```

最后，实现关闭文件的系统调用（并在 `trap_handler` 中添加相应入口），代码如下：

```

1. // arch/riscv/kernel/syscall.c
2.
3. int64_t sys_close(int fd) {
4.     int64_t ret;
5.     struct file* target_file = &(current->files[fd]);
6.     if (target_file->opened) {
7.         target_file->opened = 0;
8.         ret = 0;
9.     } else {
10.         printk("file not open\n");
11.         ret = ERROR_FILE_NOT_OPEN;
12.     }
13.     return ret;
14. }

```

3. 实现读、写文件的函数

大致遵循“找到要读写位置在物理内存中所处的簇位置→将文件内容读进缓冲→把缓冲中的数据输出于屏幕（读），或往缓冲中写入数据再将缓冲写回原位（写）”的步骤，实现如下：

```

1. int64_t fat32_read(struct file* file, void* buf, uint64_t len) {
2.     uint32_t file_size = get_file_size(file);
3.     uint64_t read_len = 0;
4.     while (read_len < len && file->cfo < file_size) {
5.         uint32_t cluster = file->fat32_file.cluster;
6.         for (uint32_t clusteri = 0;

```

```

clusteri < file->cfo / (fat32_volume.sec_per_cluster * VIRTIO_BLK_SECTOR_SIZE) && cluster < 0x0FFFFFF8; ++clusteri) {
7.         //cluster += 1;
8.         cluster = next_cluster(cluster);
9.     }
10.    uint64_t sector = cluster_to_sector(cluster);
11.    uint64_t offset_in_sector = file->cfo % VIRTIO_BLK_SECTOR_SIZE;
12.    uint64_t remain_readable_size = VIRTIO_BLK_SECTOR_SIZE - offset_in_sector;
13.    if (remain_readable_size > len - read_len)
14.        remain_readable_size = len - read_len;
15.    if (remain_readable_size > file_size - file->cfo)
16.        remain_readable_size = file_size - file->cfo;
17.    virtio_blk_read_sector(sector, fat32_buf);
18.    memset(buf, 0, len - read_len);
19.    memcpy(buf, fat32_buf + offset_in_sector, remain_readable_size);
20.
21.    file->cfo += remain_readable_size;
22.    buf = (char *)buf + remain_readable_size;
23.    read_len += remain_readable_size;
24.
25. }
26. return read_len;
27. /* todo: read content to buf, and return read length */
28.}
29.
30.int64_t fat32_write(struct file* file, const void* buf, uint64_t len) {
31.    uint64_t write_len = 0;
32.    while (len > 0) {
33.        uint32_t cluster = file->fat32_file.cluster;
34.        for (uint32_t clusteri = 0; clusteri < file->cfo / (fat32_volume.sec_per_cluster * VIRTIO_BLK_SECTOR_SIZE) && cluster < 0x0FFFFFF8; ++clusteri) {
35.            cluster = next_cluster(cluster);
36.        }
37.        uint64_t sector = cluster_to_sector(cluster);
38.        uint64_t offset_in_sector = file->cfo % VIRTIO_BLK_SECTOR_SIZE;
39.        uint64_t remain_writable_size = VIRTIO_BLK_SECTOR_SIZE - offset_in_sector;
40.        if (remain_writable_size > len) {
41.            remain_writable_size = len;
42.        }
43.        virtio_blk_read_sector(sector, fat32_buf);
44.        memcpy(fat32_buf + offset_in_sector, buf, remain_writable_size);
45.        virtio_blk_write_sector(sector, fat32_buf);
46.
47.        file->cfo += remain_writable_size;

```

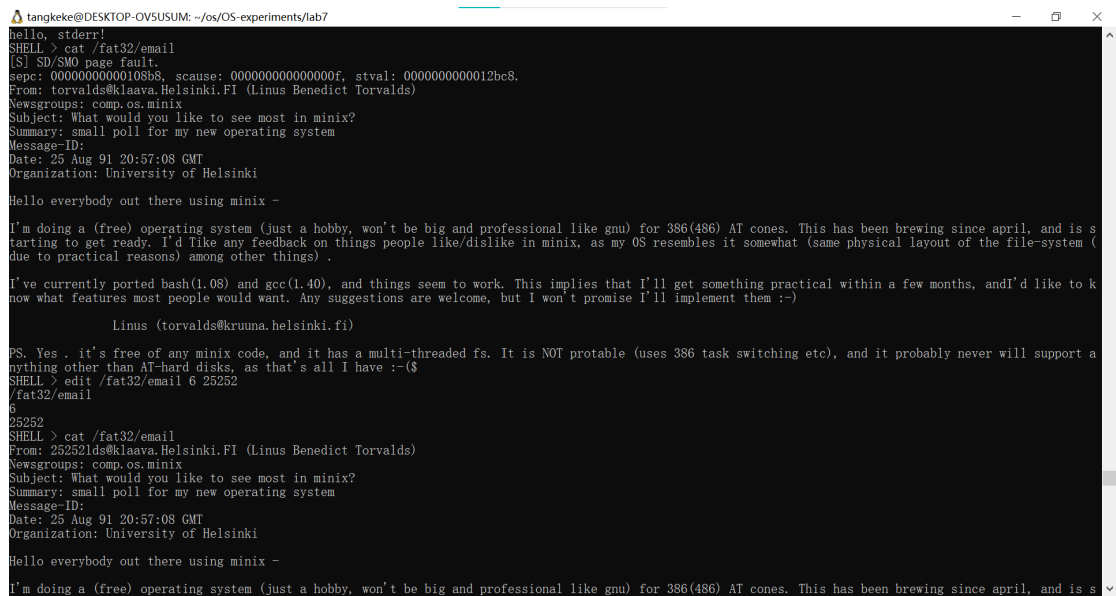
```

48.         buf += remain_writable_size;
49.         len -= remain_writable_size;
50.         write_len += remain_writable_size;
51.     }
52.     return write_len;
53.     /* todo: fat32_write */
54. }

```

(三) 编译运行

结果如下两图，表明正确读写：



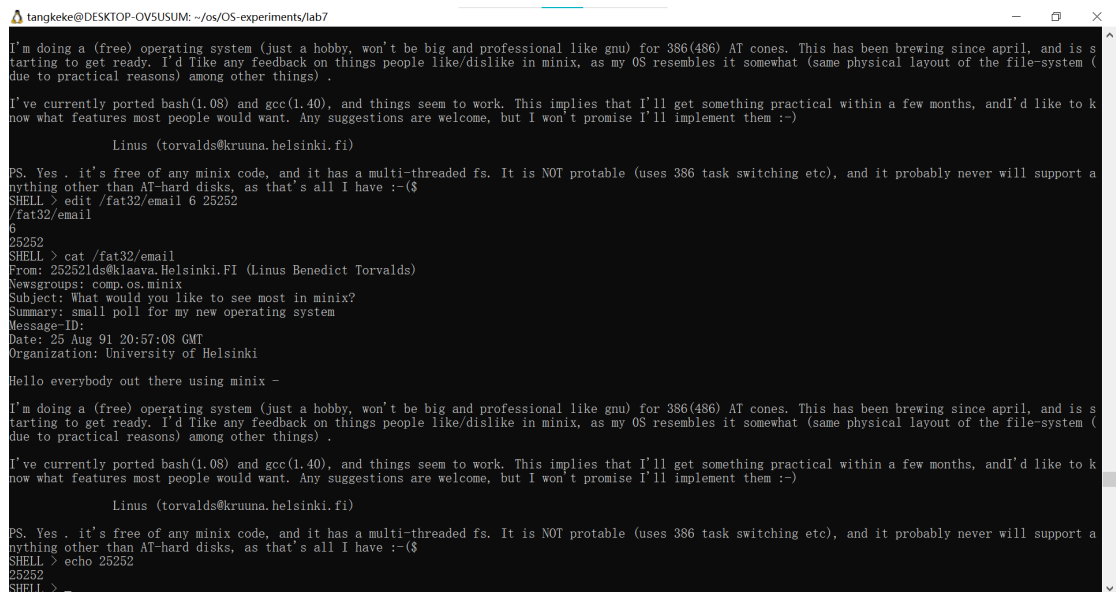
```

tangkeke@DESKTOP-OV5USUM: ~/os/os-experiments/lab7
hello, stderr!
SHELL > cat /fat32/email
[S] SD/SMD page fault.
sepc: 00000000000108b8, scause: 000000000000000f, stval: 0000000000012bc8.
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT cones. This has been brewing since april, and is s
tarting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (
due to practical reasons) among other things) .

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to k
now what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
```



```

tangkeke@DESKTOP-OV5USUM: ~/os/os-experiments/lab7

PS. Yes . it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support a
nything other than AT-hard disks, as that's all I have :-($
SHELL > edit /fat32/email 6 25252
/fat32/email
6
25252
SHELL > cat /fat32/email
From: 25252lds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT cones. This has been brewing since april, and is s
tarting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (
due to practical reasons) among other things) .

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to k
now what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
```

三、讨论和心得

这次实验大大加深了我对 FAT32 文件系统结构的理解，对簇、扇区等概念有了更多认识。为了搞明白原理以编写代码，我搜索了网上的一些比较通俗易懂的解释，并对着解释不断推测代码已经给出的数据结构中各变量的含义，并根据推测出的含义去运用它们。随着代码的不断编写、程序的不断调试，FAT32 系统结构的图景在我脑中越发明晰了！

四、思考题

无。

五、附录

参考了以下两个链接中的解释：

<https://blog.csdn.net/ZCShouCSDN/article/details/97610903>

<https://codeantenna.com/a/ubkEdR29X#:~:text=%E5%AD%A8%E9%A1%BA%E5%BA%8F%E6%98%AF%E6%96%87%E4%BB%B6%E5%90%8D1%2B%E6%96%87%E4%BB%B6%E5%90%8D2%EF%BC%8C%E5%86%85%E9%83%A8%E6%8C%89%E7%85%A7%E2%91%A0%E2%91%A1%E2%91%A2%E7%9A%84%E9%A1%BA%E5%BA%8F%EF%BC%8C%E5%A6%82%E4%B8%8B%E5%9B%BE%E6%89%80%E7%A4%BA%EF%BC%9A%20FAT%E5%83%8F%E4%B8%80%E4%B8%AA%E5%B7%A8%E5%A4%A7%E7%9A%84%E9%93%BE%E8%A1%A8%EF%BC%8C%E6%AF%8F%E4%B8%AA%E8%A1%A8%E9%A1%B9%E5%AD%97%E8%8A%82%E3%80%82,%E5%9C%A8%E7%9B%AE%E5%BD%95%E4%B8%AD%E6%89%BE%E5%88%B0%E4%B8%80%E4%B8%AA%E6%96%87%E4%BB%B6%E7%9A%84%E5%BC%80%E5%A7%8B%E7%B0%87%E5%8F%B7%EF%BC%8C%E6%9D%A5%E6%9F%A5FAT%E8%A1%A8%EF%BC%8C%E6%89%BE%E5%88%B0%E8%Bf%99%E4%B8%AA%E7%B0%87%E5%8F%B7%E5%AF%B9%E5%BA%94%E7%9A%84FAT%E8%A1%A8%E9%A1%B9%E3%80%82%20%E5%A6%82%E6%9E%9C%E8%AF%A5%E8%A1%A8%E9%A1%B9%E6%98%AF0x0FFFFFFF%EF%BC%8C%E8%A1%A8%E7%A4%BA%E6%98%AF%E8%AF%A5%E6%96%87%E4%BB%B6%E7%9A%84%E6%9C%80%E5%90%8E%E4%B8%80%E7%B0%87%EF%BC%8C%E5%90%A6%E5%88%99%E8%AF%A5%E8%A1%A8%E9%A1%B9%E6%8C%87%E5%90%91%E7%9A%84%E6%98%AF%E8%AF%A5%E6%96%87%E4%BB%B6%E5%8D%A0%E7%94%A8%E7%9A%84%E4%B8%8B%E4%B8%80%E4%B8%AA%E7%B0%87%E5%8F%B7%E3%80%82>

（虽然里面也有具体的代码实现，但和我们的实验差异巨大，想抄也是不可能的嘿嘿）