操作系统实验报告

课程名称	:	操作系统
姓　　名	:	沈一芃
学　　院	:	计算机学院
专　　业	:	计算机科学与技术
学　　号	:	3220101827
指导教师	:	李环 柳晴

2024 年　12　月　26　日

Lab 6 Report

实验名称：VFS & FAT32 文件系统

联系方式：1669335639@qq.com 18310211513

1. 实验目的与要求

1.1. 实验目的

- 了解VFS和FAT32文件系统
- 实现基础的文件读写功能

1.2. 实验要求

- 为用户态的 Shell 提供 `read` 和 `write` syscall 的实现（完成该部分的所有实现方得 60 分）
- 实现 FAT32 文件系统的基本功能，并对其中的文件进行读写（完成该部分的所有实现方得 40 分）

2. 实验环境

本机CPU	本机操作系统	实验环境	实验操作系统	配置环境
Mac Silicon M3	MacOS Sequoia 15.0.1	Docker 4.33.0	Ubuntu 24.10	Qemu-system-riscv64 9.0.2

3. 实验过程

3.1. 实验环境搭建

- 整个实验建立在lab4 or lab5的基础上。从仓库克隆实验6代码到本地，跟随实验流程搭建文件结构如下：

```
├── arch
│   └── riscv
│       ├── include
│       │   ├── mm.h
│       │   ├── proc.h
│       │   ├── clock.h
│       │   ├── syscall.h
│       │   ├── defs.h
│       │   └── sbi.h
│       ├── kernel
```

```
│       │   ├── vm.c
│       │   ├── mm.c
│       │   ├── proc.c      # need to modify; only init 1 task. add struct file
│       │   ├── syscall.c   # need to accomplish; implement sys_write and sys_read
│       │   ├── sbi.c
│       │   ├── trap.c      # need to accomplish; enable and handle new syscall
│       │   ├── clock.c
│       │   ├── head.S
│       │   ├── entry.S
│       │   ├── Makefile
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── elf.h
│   ├── printk.h
│   ├── stddef.h
│   ├── stdint.h
│   ├── stdlib.h
│   ├── fat32.h             # FAT32 相关数据结构与函数声明
│   ├── fs.h                # 供系统内核使用的文件系统相关数据结构及函数声明
│   ├── mbr.h               # MBR 相关数据结构与函数声明（无需关注）
│   ├── vfs.h               # VFS 操作函数声明
│   ├── virtio.h            # VirtIO 驱动相关数据结构与函数声明（无需关注）
│   └── string.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
├── fs
│   ├── Makefile
│   ├── fat32.c             # FAT32 文件系统实现
│   ├── fs.c                # need to accomplish;
│   ├── mbr.c               # MBR 初始化（无需修改）
│   ├── vfs.c               # VFS 实现
│   └── virtio.c            # VirtIO 驱动（无需修改）
├── user
│   ├── Makefile            # need to modify; allow PFH and FORK TEST
│   ├── main.c              # need to modify; directly pull from storage. Need to read!
important!
│   ├── link.lds
│   ├── printf.c
│   ├── start.S
│   ├── stddef.h
│   ├── stdio.h
│   ├── string.c            # need to modify; directly pull from storage
│   ├── string.h            # need to modify; directly pull from storage
│   ├── syscall.h
│   ├── unistd.c            # need to modify; directly pull from storage
│   ├── unistd.h            # need to modify; directly pull from storage
```

```
│   └── uapp.S
├── disk.img
└── Makefile
```

- 本次实验时间较为匆忙，上述文件结构可能有部分不清晰。且本实验仅完成了第一部分。

3.2. Shell: 与内核进行交互

3.2.0. 总括

- 本次实验模拟了文件系统的管理。过去的`sys_write`我们直接进行操作就好。本次实验则是通过调用文件来实现。所谓的调用文件，其实就是根据struct file中记录的相关信息来对文件做管理，查看不同信息情况下应该做什么操作。
  - 故而，与文件系统相关的部分就是读懂struct file的结构，对其初始化。
  - 其余部分都是完善一些系统调用等功能性函数。

3.2.1. 文件系统抽象与初始化

- 文件系统的结构体定义如下：
  - 基本信息包括文件是否打开、文件权限、指针偏移、文件类型。进阶操作包含三个函数指针，使能该文件的不同功能。

```
struct file {   // Opened file in a thread.
    uint32_t opened;
    uint32_t perms;
    int64_t cfo;
    uint32_t fs_type;

    union {
        struct fat32_file fat32_file;
    };

    int64_t (*lseek) (struct file *file, int64_t offset, uint64_t whence);
    int64_t (*write) (struct file *file, const void *buf, uint64_t len);
    int64_t (*read)  (struct file *file, void *buf, uint64_t len);

    char path[MAX_PATH_LENGTH];
};
```

- 在`init_task`中，我们要为线程结构体加入文件系统，并对其做初始化。
  - 这里对于vma的处理，是申请一个全新的页。显然，这样十分浪费内存。可以在buddy system中自行修改，添加一个小块内

```
void task_init() {
    srand(2024);
    nr_tasks = 1 + 1;
    for (int i = 1; i < nr_tasks; i++){
        task[i] = kalloc();
        ...
        // assign information, init pgd, init vma .etc
        task[i]->files = file_init();
        printk("after task[%lu] initialize\n", i);
    }
```

- 文件系统初始化：
  - 分配内存，若担心浪费，同样可以`buddy_system`中自定义接口。此处直接分配整页
  - 由于我们只对0/stdin，1/stdout，2/stderr三个文件做管理，只需初始化这三个结构体
  - 写入对应的权限、打开状态，更新函数指针，赋予其特定功能

```
struct files_struct *file_init() {
    struct files_struct *ret = alloc_page();
    ret->fd_array[0].opened = 1;
    ret->fd_array[0].perms = FILE_READABLE;
    ret->fd_array[0].cfo = 0;
    ret->fd_array[0].lseek = NULL;
    ret->fd_array[0].write = NULL;
    ret->fd_array[0].read = stdin_read;
    ret->fd_array[1].opened = 1;
    ret->fd_array[1].perms = FILE_WRITABLE;
    ret->fd_array[1].cfo = 0;
    ret->fd_array[1].lseek = NULL;
    ret->fd_array[1].write = stdout_write;
    ret->fd_array[1].read = NULL;
    ret->fd_array[2].opened = 1;
    ret->fd_array[2].perms = FILE_WRITABLE;
    ret->fd_array[2].cfo = 0;
    ret->fd_array[2].lseek = NULL;
    ret->fd_array[2].write = stderr_write;
    ret->fd_array[2].read = NULL;

    return ret;
}
```

3.2.2. 处理stdout/stderr的写入

- 系统调用`sys_write`会要求某个文件做相应操作。此处是处理stdout和stderr。
  - 捕获中断后，在`trap_handler`中添加识别`sys_write`系统调用，在对应的函数中根据传来的参数，利用文件系统进行管理。
- 根据文件类型，查询操作权限，调用相关操作。

```
uint64_t sys_write(unsigned int fd, const char* buf, uint64_t len){
    int64_t ret;
    struct file *file = &(current->files->fd_array[fd]);
    if (file->opened == 0) {
        printk("file not opened\n");
        return ERROR_FILE_NOT_OPEN;
    } else {
        if (file->perms == FILE_WRITABLE){
            file->write(file, buf, len);
        }
    }
    return ret;
}
```

3.2.3. 处理 stdin 的读取

- 操作同理，系统调用`sys_read`会要求某个文件做相应操作。此处是处理stdout和stderr。
  - 捕获中断后，在`trap_handler`中添加识别`sys_read`系统调用，在对应的函数中根据传来的参数，利用文件系统进行管理。
- 根据文件类型，查询操作权限，调用相关操作。

```
uint64_t sys_read(unsigned int fd, const char* buf, uint64_t len){
    int64_t ret;
    struct file *file = &(current->files->fd_array[fd]);
    if (file->opened == 0) {
        printk("file not opened\n");
        return ERROR_FILE_NOT_OPEN;
    } else {
        if ((fd == 1 || fd == 2) && file->perms == FILE_WRITABLE){
            file->write(file, buf, len);
        } else if (fd == 0 && file->perms == FILE_READABLE){
            ret = file->read(file, buf, len);
        }
    }
    return ret;
}
```

- 然而，我们需要处理终端的输入，即完成工具性系统调用`sbi_debug_console_read`。

```
struct sbiret sbi_debug_console_read(uint64_t num_bytes, uint64_t base_addr_lo,
uint64_t base_addr_hi){
    return sbi_ecall(0x4442434E, 0x1, num_bytes, base_addr_lo, base_addr_hi, 0,0,0);
}
```

> 此处要特别注意文档中提及的，`ecall`的实现。正确的寄存器绑定方式如下：
>
> ```
> struct sbiret sbi_ecall(uint64_t eid, uint64_t fid,
> ```

```
                        uint64_t arg0, uint64_t arg1, uint64_t arg2,
                        uint64_t arg3, uint64_t arg4, uint64_t arg5) {
    struct sbiret res;
    long error;
    long value;
    asm volatile (
        "mv a7, %[eid]\n"
        "mv a6, %[fid]\n"
        "mv a0, %[arg0]\n"
        "mv a1, %[arg1]\n"
        "mv a2, %[arg2]\n"
        "mv a3, %[arg3]\n"
        "mv a4, %[arg4]\n"
        "mv a5, %[arg5]\n"
        "ecall\n"
        "mv %[error], a0\n"
        "mv %[value], a1\n"
        : [error] "=r"(error),[value] "=r"(value)
        : [eid] "r"(eid), [fid] "r"(fid), [arg0] "r"(arg0), [arg1] "r"(arg1),
[arg2] "r"(arg2), [arg3] "r"(arg3), [arg4] "r"(arg4), [arg5] "r"(arg5)
        : "memory", "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
    );
    res.error = error;
    res.value = value;
    return res;
}
```

3.2.4 测试

- 至此，我们实现了nish的读写功能

```
entry: 100e8, memsz: 4308, filesz: 226c
after task[1] initialize
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [pid = 1, priority = 7, priority = 7]
[trap.c,88,do_page_fault] [pid = 1, PC = 100e8], valid page fault at '100e8' with scause c

[vm.c,149,create_mapping] root: ffffffe0002d4000, paddr: [802d8000, 802d9000), vaddr: [10000, 11000), perm: 1f

[trap.c,88,do_page_fault] [pid = 1, PC = 10aa0], valid page fault at '3fffffff8' with scause f

[vm.c,149,create_mapping] root: ffffffe0002d4000, paddr: [802db000, 802dc000), vaddr: [3fffff000, 4000000000), perm: 17

[trap.c,88,do_page_fault] [pid = 1, PC = 11c1c], valid page fault at '11c1c' with scause c

[vm.c,149,create_mapping] root: ffffffe0002d4000, paddr: [802de000, 802df000), vaddr: [11000, 12000), perm: 1f

hello, stdout!
hello, stderr!
[trap.c,88,do_page_fault] [pid = 1, PC = 10c04], valid page fault at '14000' with scause d

[vm.c,149,create_mapping] root: ffffffe0002d4000, paddr: [802df000, 802e0000), vaddr: [14000, 15000), perm: 1f

SHELL > echo "test English"
[trap.c,88,do_page_fault] [pid = 1, PC = 102dc], valid page fault at '13000' with scause f

[vm.c,149,create_mapping] root: ffffffe0002d4000, paddr: [802e0000, 802e1000), vaddr: [13000, 14000), perm: 1f

test English
SHELL > echo SET [PID = 1 PRIORITY = 7 COUNTER = 7]
"test $^&^&*()SET [PID = 1 PRIORITY = 7 COUNTER = 7]
:"
test $^&*^&*():
SHELL > echo "test 中文"
test 中文
SHELL > SET [PID = 1 PRIORITY = 7 COUNTER = 7]
echo "ZJU OSET [PID = 1 PRIORITY = 7 COUNTER = 7]
perating System 2024 Final"
ZJU Operating System 2024 Final
SHELL >
```

4. 讨论与心得

- 本次实验的前一部分，也就是我实现的部分较为简单，只要理清了文件系统的作用，整个思路就会通畅很多。这个实验就是将一些操作改用文件系统来实现，而管理、调用文件系统则依靠`struct file`结构体。之后的内容只需要跟着逻辑链条一层层函数跳转就好，本质就是系统调用的功能性实现。
- 较为感慨，os的实验做到了最后一个要画上句号。这门课程的实验设计让我受益良多。我遇到过很多问题，不仅仅是技术实现上的，还有关于我们做的这个theoratical toy的漏洞，以及和真正的复杂系统的联系。但种种实现上的漏洞和奇怪的思考都在助教的帮助下得到了解答，真的很感恩老师和助教们一学期的辛苦付出。最后一个实验bonus做得很顺利，比之前的每一个实验都要顺利，算是画上了一个不错的句号。只是可惜由于自己的拖延症，期末没时间把更难的PART2做完。希望以后有机会，能补完这个实验。如果有机会，也愿意来当助教，或者只是contributor，帮课程组一起完善这门实验，**MAKE ZJU-OS GREAT AGAIN**!