操作系统lab6：VFS & FAT32文件系统

教师：李环、柳晴
学号：3220104519
姓名：蔡佳伟

一、实验目的

- 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）

二、实验过程

2.1 准备工程

同步文件后：

修改根目录下Makefile

修改log宏 并添加相关编译产物以编译新加入的fs文件夹下的内容

```
INCLUDE :=  -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF      :=  -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-pie -fno-builtin -ffuncti
LOG     :=  1
CFLAG   :=  $(CF) $(INCLUDE) -DTEST_SCHED=$(TEST_SCHED) -DLOG=$(LOG)
```

```
all: clean
    $(MAKE) -C lib all
    $(MAKE) -C user all
    $(MAKE) -C init all
    $(MAKE) -C fs all
    $(MAKE) -C arch/riscv all
    @echo -e '\n'Build Finished OK

run: all
    @echo Launch qemu...
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default

debug: all
    @echo Launch qemu for debug...
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s

clean:
    $(MAKE) -C lib clean
    $(MAKE) -C init clean
    $(MAKE) -C arch/riscv clean
    $(MAKE) -C fs clean
    $(MAKE) -C user clean
    $(shell test -f vmlinux && rm vmlinux)
    $(shell test -f vmlinux.asm && rm vmlinux.asm)
    $(shell test -f System.map && rm System.map)
    @echo -e '\n'Clean Finished
```

修改LOG相关的宏

2.2 Shell: 与内核进行交互

修改proc.h: 为task_struct结构体添加一个files指针指向文件表

```
/* 线程数据结构 */
struct task_struct
{
    uint64_t state;    // 线程状态 0
    uint64_t counter;  // 运行剩余时间 8
    uint64_t priority; // 运行优先级 1 最低 10 最高, 16
    uint64_t pid;      // 线程 id 24

    struct thread_struct thread;
    uint64_t *pgd;       // 用户态页表 168
    struct mm_struct mm; // 176

    struct files_struct *files;
};
```

修改fs.c中file_init()函数:

因为files_struct含有MAX_FILE_NUMBER=16个file结构体，所有只需要一个页面空间

```
struct files_struct *file_init() {
    struct files_struct *ret = (struct files_struct *)alloc_page(1);
    memset(ret, 0, PGSIZE);

    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].read = stdin_read;
    ret->fd_array[0].write = NULL;
    memcpy(ret->fd_array[0].path, "stdin", 6);

    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].read = NULL;
    ret->fd_array[1].write = stdout_write;
    memcpy(ret->fd_array[1].path, "stdout", 7);

    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].read = NULL;
    ret->fd_array[2].write = stderr_write;
    memcpy(ret->fd_array[2].path, "stderr", 7);
    return ret;
}
```

在task_init()中调用file_init()函数

```
// set_task_pgd(task[i]);
load_program(task[i]);
task[i]->thread.sstatus = ((1 << 18) | (1 << 5)); // SUM SPIE
task[i]->thread.sscratch = (USER_END);

printk("task[%d] pid = %d priority = %d\n", i, task[i]->pid, task[i]->priority);

task[i]->files = file_init();
```

修改sys_write()函数：

```
// 写入系统调用
int sys_write(unsigned int fd, const char *buf, size_t count)
{
    printk("enter sys_write\n");
    // if (fd != 1)
    // {
    //     return -1;  // 如果文件描述符不是标准输出，返回错误
    // }
    // for (size_t i = 0; i < count; i++)
    // {
    //     printk("%c", buf[i]);  // 打印每个字符到控制台
    // }
    // return count;  // 返回写入的字符数

    int64_t ret;
    struct file *file = &(current->files->fd_array[fd]);

    if(file->opened==0){
        printk("file %d not open\n", fd);
        return ERROR_FILE_NOT_OPEN;
    }
    else if(!(file->perms & FILE_WRITABLE) || file->write == NULL){
        printk("file %d not write\n", fd);
        return ERROR_FILE_NOT_OPEN;
    }

    return file->write(file, buf, count);
}
```

并且补充sys_read()函数：

```c
int sys_read(unsigned int fd, const char *buf, size_t count)
{
    printk("enter sys_read\n");
    int64_t ret;
    struct file *file = &(current->files->fd_array[fd]);

    if(file->opened==0){
        printk("file %d not open\n", fd);
        return ERROR_FILE_NOT_OPEN;
    }
    else if(!(file->perms & FILE_WRITABLE) || file->write == NULL){
        printk("file %d not write\n", fd);
        return ERROR_FILE_NOT_OPEN;
    }

    return file->read(file, buf, count);
}
```

完成fs/vfs.c中的stdin_read()和stderr_write()函数：

```c
int64_t stdin_read(struct file *file, void *buf, uint64_t len) {
    for (int i = 0; i < len; i++) {
        ((char *)buf)[i] = uart_getchar();
    }
    return len;
}

int64_t stdout_write(struct file *file, const void *buf, uint64_t len) {
    char to_print[len + 1];
    for (int i = 0; i < len; i++) {
        to_print[i] = ((const char *)buf)[i];
    }
    to_print[len] = 0;
    return printk(to_print);
}

int64_t stderr_write(struct file *file, const void *buf, uint64_t len) {
    char to_print[len + 1];
    for (int i = 0; i < len; i++) {
        to_print[i] = ((const char *)buf)[i];
    }
    to_print[len] = 0;
    return printk(to_print);
}
```

在中断处理中添加read的处理：

```c
case SYS_WRITE:
    regs->x[10] = sys_write((unsigned int)regs->x[10], (const char *)regs->x[11], (size_t)regs->x[12]);  // 处理write系统调用
    break;
case SYS_GETPID:
    regs->x[10] = sys_getpid();  // 处理getpid系统调用
    break;
case SYS_CLONE:
    regs->x[10] = do_fork(regs);  // 处理fork系统调用
    break;
case SYS_READ:
    regs->x[10] = sys_read((unsigned int)regs->x[10], (const char *)regs->x[11], (size_t)regs->x[12]);
```

补充sbi.c中的函数：并在sbi.h中声明

```
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);
}
```

使用make run LOG=0进行测试：

实现了输入和输出：

```
2024 ZJU Operating System
hello, stdout!
hello, stderr!
SHELL > echo hello
hello
SHELL > echo cjw
cjw
SHELL >
```

三、讨论和心得

在过程中，遇到了很多问题，比如mv内联汇编的问题就遇到了，于是修改代码：

```
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 ret;
    uint64_t error, 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"
        : "=r"(error), "=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)
        : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7", "memory");
    ret.error = error;
    ret.value = value;
    return ret;
}
```

这样编译器就不会使用这些寄存器作为临时变量

在文件中还运用了string.h中的函数，于是在string.c中补全，并在string.h中声明：

```c
uint64_t strlen(const void *str){
    uint64_t length = 0;
    const char *cstr = (char *)str;
    while(cstr[length] != '\0'){
        length++;
    }
    return length;
}

int memcmp(const void *ptr1, const void *ptr2, uint64_t n){
    const unsigned char *p1 = ptr1;
    const unsigned char *p2 = ptr2;
    for(uint64_t i = 0; i < n; i++){
        if(p1[i] != p2[i]){
            return (p1[i] < p2[i]) ? -1 : 1;
        }
    }
    return 0;
}
```

还有为proc.h加入#include "fs.h"

还有在两个syscall.h文件中补充SYS_READ的声明等

在看报错的过程中，有哪个地方错误就随手改掉了，不过改的过多了有一次，突然提示找不到头文件，但是头文件明明都有，实在没有办法了，之前也遇到过一次这个bug，于是就重来了一遍，不过好在这次没有遇到。

据室友说，是调用函数没有使用extern声明的原因，但是我不知道具体是哪里，也没有排查出来是哪一次不小心改掉了，于是就算了。

四、思考题

这次没有思考题，oh yeah