浙江大学

《操作系统原理与实践》
实验报告

实验名称	：	Lab6: VFS & FAT32 文件系统
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	申文博
助　　教	：	陈淦豪&王鹤翔&许昊瑞

2024 年 12 月 28 日

Lab6: VFS & FAT32 文件系统

Lab6：VFS & FAT32 文件系统

1 实验内容及简要原理介绍

- 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）

虚拟文件系统（Virtual File System，VFS）或虚拟文件系统交换机是位于更具体的文件系统之上的抽象层。VFS 的目的是允许客户端应用程序以统一的方式访问不同类型的具体文件系统。例如，可以使用 VFS 透明地访问本地和网络存储设备，而客户机应用程序不会注意到其中的差异。它可以用来弥合 Windows、macOS 等不同操作系统所使用的文件系统之间的差异，这样应用程序就可以访问这些类型的本地文件系统上的文件，而不必知道它们正在访问什么类型的文件系统。

VFS 指定内核和具体文件系统之间的接口（或“协议”）。因此，只需完成协议，就可以很容易地向内核添加对新文件系统类型的支持。协议可能会随着版本的不同而不兼容地改变，这将需要重新编译具体的文件系统支持，并且可能在重新编译之前进行修改，以允许它与新版本的操作系统一起工作；或者操作系统的供应商可能只对协议进行向后兼容的更改，以便为操作系统的给定版本构建的具体文件系统支持将与操作系统的未来版本一起工作。

2 实验具体过程与代码实现

本次实验的内容这里只完成一个部分：

- Shell：实现 VFS，编写虚拟文件设备 stdin，stdout，stderr 的读写操作；

实验依赖于Lab5实现环境

2.1 Shell：与内核进行交互

需要修改 proc.h/c，在初始化时只创建一个用户态进程

```
//added in Lab5

uint64_t nr_tasks = 2;              // 当前线程数量(含IDLE线程)

struct vm_area_struct *find_vma(struct mm_struct *mm, uint64_t addr
    for(struct vm_area_struct *vma = mm->mmap; vma; vma = vma->vm_n
        if(vma->vm_start <= addr && addr < vma->vm_end){
            return vma;
```

因为加了一个根目录下的 `fs` 文件夹，所以需要在 `arch/riscv/Makefile` 里面添加相关编译产物来进行链接：

```
  ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o
../../lib/*.o ../../fs/*.o ../../user/uapp.o -o ../../vmlinux
```

src > lab6 > arch > riscv > Makefile

```
all:
    ${MAKE} -C kernel all
    ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../fs/*.o ../../user/uapp.o -o ../../vmlinux
    $(shell test -d boot || mkdir -p boot)
    ${OBJCOPY} -O binary ../../vmlinux ./boot/Image
    ${OBJDUMP} -S ../../vmlinux > ../../vmlinux.asm
    nm ../../vmlinux >  ../../System.map

clean:
    ${MAKE} -C kernel clean
    $(shell test -d boot && rm -rf boot)
```

此外我们也需要修改根目录下的`Makefile`,加入`fs`文件夹的编译

src > lab6 > Makefile

```
## added by lab6
LOG     := 1
## __added by lab6

.PHONY:all run debug clean
all: clean
    $(MAKE) -C lib all
    $(MAKE) -C fs all
    $(MAKE) -C init all
    $(MAKE) -C user 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 fs clean
    $(MAKE) -C init clean
    $(MAKE) -C user clean
    $(MAKE) -C arch/riscv 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
```

2.1.1 文件系统抽象

修改 `proc.h`，为进程 `task_struct` 结构体添加一个指向文件表的指针：

src > lab6 > arch > riscv > include > C proc.h > NR_TASKS

```c
#ifndef __PROC_H__
struct task_struct {
    uint64_t state;      // 线程状态
    uint64_t counter;    // 运行剩余时间
    uint64_t priority;   // 运行优先级 1 最低 10 最高
    uint64_t pid;        // 线程 id

    struct thread_struct thread;
    //added in Lab4
    //必须加在thread_struct后面，因为在__switch_to函数中会计算thread变量的位置
    uint64_t *pgd;  // 用户态页表
    //end added in Lab4

    //added in Lab5
    struct mm_struct mm;
    //end added in Lab5

    // added in Lab6
    struct files_struct *files;
    // end added in Lab6
};

//added in Lab5
```

2.1.2 stdout/err/in 初始化

定义了一个函数 `file_init`：

src > lab6 > fs > C fs.c > ...

```c
#include "fs.h"

struct files_struct *file_init() {
    // todo: alloc pages for files_struct, and initialize stdin, stdout, stderr
    struct files_struct *ret = (struct files_struct *)alloc_pages(PGROUNDUP(sizeof(struct files_struct)));
    memset(ret,0,sizeof(PGROUNDUP((sizeof(struct files_struct)))));
    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 = (int64_t*)stdin_read;
    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].write = (int64_t*)stdout_write;
    ret->fd_array[1].read = NULL;
    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].write = (int64_t*)stderr_write;
    ret->fd_array[2].read = NULL;
    memcpy(ret->fd_array[2].path,"stderr",7);
    return ret;
}
```

此外 `fs.c` 实现中有用到字符串相关函数，我们给予补充：

src > lab6 > include > C string.h > memcmp(const void *, const void *, uint64_t)

```h
#ifndef __STRING_H__
#define __STRING_H__

#include "stdint.h"

void *memset(void *, int, uint64_t);

void *memcpy(void *, void *, uint64_t);

int memcmp(const void *, const void *, uint64_t);

int strlen(const char *);

#endif
```

src > lab6 > lib > C string.c > memcmp(const void *, const void *, uint64_t)

```c
#include "string.h"
#include "stdint.h"

void *memset(void *dest, int c, uint64_t n) {
    char *s = (char *)dest;
    for (uint64_t i = 0; i < n; ++i) {
        s[i] = c;
    }
    return dest;
}

//added in Lab4
void *memcpy(void *dest, void *src, uint64_t n) {
    char *d = dest;
    char *s = src;
    while (n--) {
        *(d++) = *(s++);
    }
    return dest;
}
//end added in Lab4

int memcmp(const void *s1, const void *s2, uint64_t n) {
    const unsigned char *c1 = s1, *c2 = s2;
    while (n-- > 0) {
        if (*c1 != *c2) {
            return *c1 - *c2;
        }
        c1++;
        c2++;
    }
    return 0;
}

int strlen(const char *s) {
    int len = 0;
    while (s[len]) {
        len++;
    }
    return len;
}
```

2.1.3 处理 stdout/err 的写入

用户态程序在开始的时候会通过 `write` 函数来向内核发起 **syscall** 进行测试。在捕获到 **write** 的 **syscall** 之后，我们就可以查找对应的 **fd**，并通过对应的 **write** 函数调用来进行输出了。一个参考实现如下：

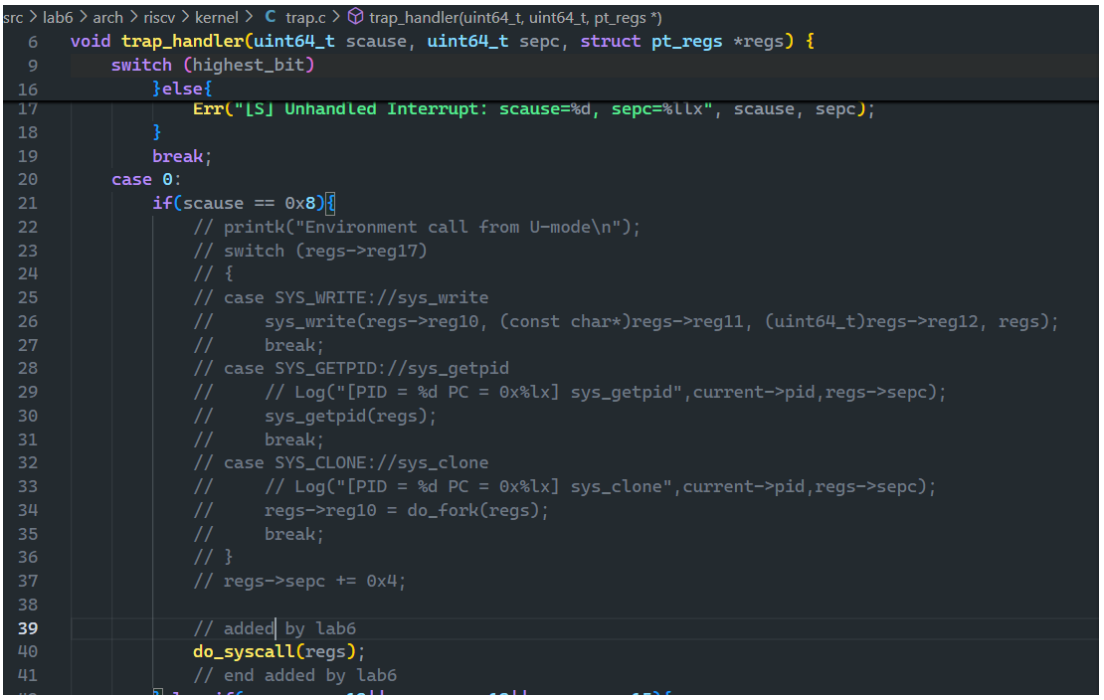
src > lab6 > arch > riscv > kernel > C syscall.c > do_syscall(pt_regs *)

```c
int64_t my_sys_write(uint64_t 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 {
        // check perms and call write function of file
        if (file->perms & FILE_WRITABLE) {
            ret = file->write(file, buf, len);
        } else {
            printk("file not writable\n");
            ret = -1;
        }
    }
    return ret;
}

int64_t my_sys_read(uint64_t fd, 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 {
        // check perms and call read function of file
        if (file->perms & FILE_READABLE) {
            ret = file->read(file, buf, len);
        } else {
            printk("file not readable\n");
            ret = -1;
        }
    }
    return ret;
}
```

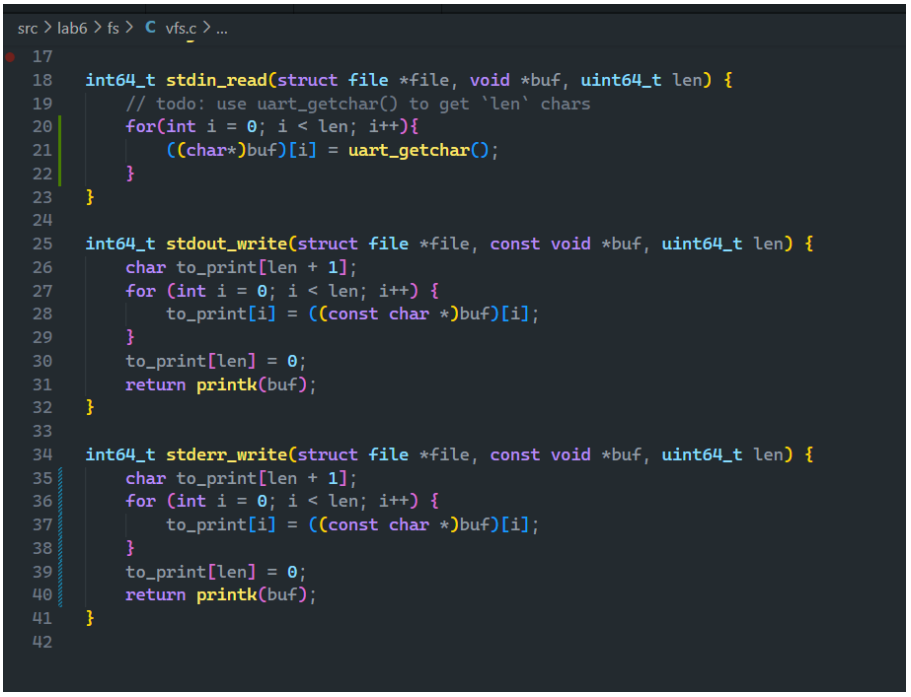
src > lab6 > arch > riscv > kernel > C syscall.c > do_syscall(pt_regs *)

```c
void do_syscall(struct pt_regs* regs) {
    switch (regs->reg17) {
        case SYS_WRITE:
            regs->reg10 = my_sys_write(regs->reg10, (const char *)regs->reg11, regs->reg12);
            break;
        case SYS_READ:
            regs->reg10 = my_sys_read(regs->reg10, (const char *)regs->reg11, regs->reg12);
            break;
        case SYS_GETPID:
            regs->reg10 = current->pid;
            break;
        case SYS_CLONE:
            regs->reg10 = do_fork(regs);
            break;
        default:
            Err("not support syscall id = %d", regs->reg7);
    }
    regs->sepc += 4;
}
```

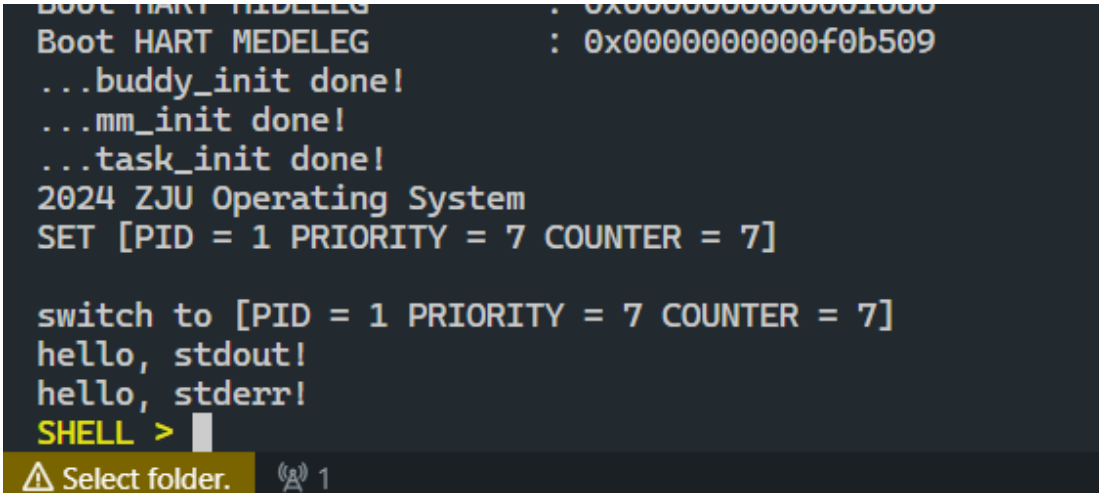
在`syscall.c`文件中我们实现了`syscall`函数的实现，重写了`syswrite`,新建了`sysread`，我们在这里需要对`trap.c`中处理**syscall**的地方做出修改,这样我们就可以利用我们的函数处理系统调用了：



对于 **stdout** 和 **stderr** 的输出，我们直接通过 `printk` 进行串口输出即可，这里的函数便是我们对应描述符调用的实际函数了：



现在我们已经可以执行输出了：

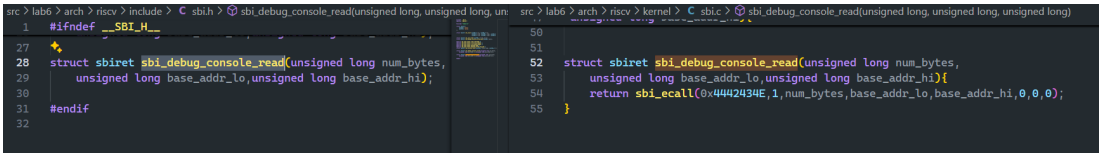


2.1.4 处理 stdin 的读取

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。

对于输入的读取就是对于 **fd=0** 的 **stdin** 文件进行 **read** 操作，所以需要我们实现 **vfs.c** 中的 **stdin_read** 函数。而对于终端的输入，我们需要通过 **sbi** 来完成，需要大家在 `arch/riscv/include/sbi.h` 中添加函数：

> 这里的实现在Lab1已经顺带实现了，这里直接贴出函数



这里我们根据实验文档指导，修复了调用**ecall**出现修改**a0-a7**的**bug**

src > lab6 > arch > riscv > kernel > C sbi.c > sbi_ecall(uint64_t, uint64_t, uint64_t, uint64_t, uint64_t, uint64_t, uint64_t, uint64_t)

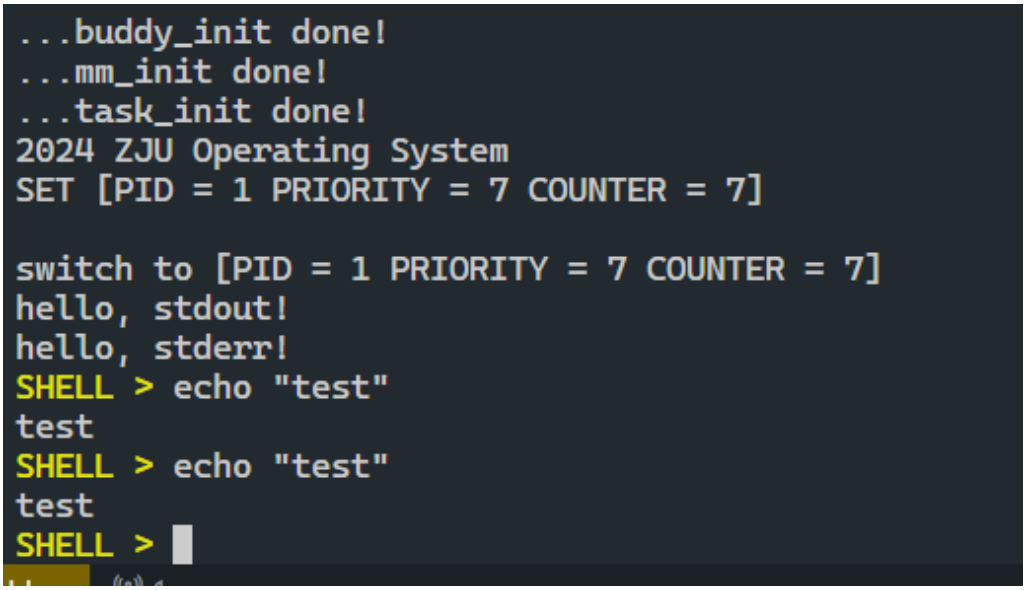
```c
#include "stdint.h"
#include "sbi.h"

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, %[ob_e]\n"
        "mv a6, %[ob_f]\n"
        "mv a0, %[ob_0]\n"
        "mv a1, %[ob_1]\n"
        "mv a2, %[ob_2]\n"
        "mv a3, %[ob_3]\n"
        "mv a4, %[ob_4]\n"
        "mv a5, %[ob_5]\n"
        "ecall\n"
        "mv %[error], a0\n"
        "mv %[value], a1\n"
        : [error] "=r" (error), [value] "=r" (value)
        : [ob_e] "r" (eid), [ob_f] "r" (fid),
          [ob_0] "r" (arg0), [ob_1] "r" (arg1),
          [ob_2] "r" (arg2), [ob_3] "r" (arg3),
          [ob_4] "r" (arg4), [ob_5] "r" (arg5)
        : "memory","a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
    );
    ret.error = error;
    ret.value = value;
    return ret;
}
```

src > lab6 > user > C printf.c > printf(const char *, ...)

```c
int printf(const char* s, ...) {
    va_list vl;
    va_start(vl, s);
    res = vprintfmt(putc, s, vl);
    long syscall_ret, fd = 1;
    buffer[tail++] = '\0';
    asm volatile ("li a7, %1\n"
                  "mv a0, %2\n"
                  "mv a1, %3\n"
                  "mv a2, %4\n"
                  "ecall\n"
                  "mv %0, a0\n"
                  : "+r" (syscall_ret)
                  : "i" (SYS_WRITE), "r" (fd), "r" (&buffer), "r" (tail)
                  : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7");
    tail = 0;
    va_end(vl);
    return res;
}
```

3 实验结果与分析

贴出`echo`命令



4　心得体会

Nice OS!