操作系统实验六

课程名称：	操作系统
作业名称：	VFS & FAT32 文件系统
姓　　名：	仇国智
学　　号：	3220102181
电子邮箱：	3220102181@zju.edu.cn
联系电话：	13714176104
指导教师：	申文博

2024 年 12 月 28 日

目录

1 实验内容

- 为用户态的 Shell 提供 read 和 write syscall 的实现 (60%)

2 实验原理

2.1 VFS

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.

3 实验具体过程与代码实现

3.1 准备工作

同步文件后在需要在根目录的 Makefile 中打开 fs 目录编译和清理, 在 arch/riscv/-Makefile 中添加链接选项:

- 根目录:

```
1    all: clean
2        $(MAKE) -C lib all
3        $(MAKE) -C user all
4        $(MAKE) -C init all
5        $(MAKE) -C fs all
6        $(MAKE) -C arch/riscv all
```

```
7         @echo -e '\n'Build Finished OK
8
9     ...
10    clean:
11        $(MAKE) -C lib clean
12        $(MAKE) -C user clean
13        $(MAKE) -C init clean
14        $(MAKE) -C fs clean
15        $(MAKE) -C arch/riscv clean
16
17        $(shell test -f vmlinux && rm vmlinux)
18        $(shell test -f vmlinux.asm && rm vmlinux.asm)
19        $(shell test -f System.map && rm System.map)
20        @echo -e '\n'Clean Finished
```

- arch/riscv:

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

我们还添加了一些工具函数不然编译不通过:

```
1   static int strlen(const char *str) {
2       int len = 0;
3       while (*str++)
4           len++;
5       return len;
6   }
7
```

```
8   static int memcmp(const void *s1, const void *s2, uint64_t n) {
9       const unsigned char *c1 = s1, *c2 = s2;
10      for (uint64_t i = 0; i < n; i++) {
11          if (c1[i] != c2[i]) {
12              return c1[i] - c2[i];
13          }
14      }
15      return 0;
16  }
```

3.2 文件系统抽象

我们使用了文件结构数组来引用文件, 实现了函数 file_init , 在 proc.c 中的 task_init 函数中为每个进程调用, 创建文件表并保存在 task struct 中. 我们在启动一个用户态程序 (包括 nish) 时默认打开了三个文件,stdin,stdout 和 stderr, 他们对应的 file descriptor 分别为 0,1,2.

```
1   struct files_struct *file_init() {
2       // todo: alloc pages for files_struct, and initialize stdin,
            stdout, stderr
3       struct files_struct *ret = (struct files_struct
            *)alloc_pages((sizeof(struct files_struct) + PGSIZE - 1) /
            PGSIZE);
4       memset(ret, 0, sizeof(struct files_struct));
5       ret->fd_array[0].opened = 1;
6       ret->fd_array[0].perms = FILE_READABLE;
7       ret->fd_array[0].cfo = 0;
8       ret->fd_array[0].lseek = NULL;
9       ret->fd_array[0].write = NULL;
10      ret->fd_array[0].read = stdin_read;
11
12      ret->fd_array[1].opened = 1;
13      ret->fd_array[1].perms = FILE_WRITABLE;
14      ret->fd_array[1].cfo = 0;
15      ret->fd_array[1].lseek = NULL;
16      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;
    }
```

3.3 处理 stdout/err 的写入

我们编写了三个函数, sys_write 用于处理用户抽象的文件描述符写入函数调用,stdout_write 和 stderr_write 用于处理 stdout 和 stderr 的写入, 他们在文件系统抽象中被初始化至文件表中.

sys_write:

```
    static uint64_t sys_write(struct pt_regs *regs)
    {
        int fd = regs->x[10];
        char *buf = (char *)regs->x[11];
        int count = regs->x[12];
        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)) {
                printk("file not writable\n");
                return ERROR_FILE_NOT_OPEN;
            }
            ret = file->write(file, buf, count);
        }
        return ret;
```

```
19      }
```

stdout_write:

```
1       int64_t stdout_write(struct file *file, const void *buf,
            uint64_t len) {
2           char to_print[len + 1];
3           for (int i = 0; i < len; i++) {
4               to_print[i] = ((const char *)buf)[i];
5           }
6           to_print[len] = 0;
7           return printk(buf);
8       }
```

stderr_write:

```
1       int64_t stderr_write(struct file *file, const void *buf,
            uint64_t len) {
2           char to_print[len + 1];
3           int64_t ret;
4           for (int i = 0; i < len; i++) {
5               to_print[i] = ((const char *)buf)[i];
6           }
7           to_print[len] = 0;
8           printk(RED);
9           ret = printk(buf);
10          printk(CLEAR);
11          return ret;
12      }
```

3.4 处理 stdin 的读取

我们在先前的实验中已经实现了 sbi_debug_console_read , 故现在只需要将其封装为 stdin_read 即可. 在 vfs.c 中定义了函数 uart_getchar(), 因为 sbi_debug_console_read 是非阻塞的, 所以我们需要一个函数来不断进行读取, 直到读到了有效字符, 然后在 stdin_read 中只需要这样读取 len 个字符就好了.

```
1       int64_t stdin_read(struct file *file, void *buf, uint64_t len) {
```

```
        // todo: use uart_getchar() to get len chars
        for (int i = 0; i < len; i++) {
            ((char *)buf)[i] = uart_getchar();
        }
        return len;
    }
```

4 实验结果与分析

我们编译运行实验:

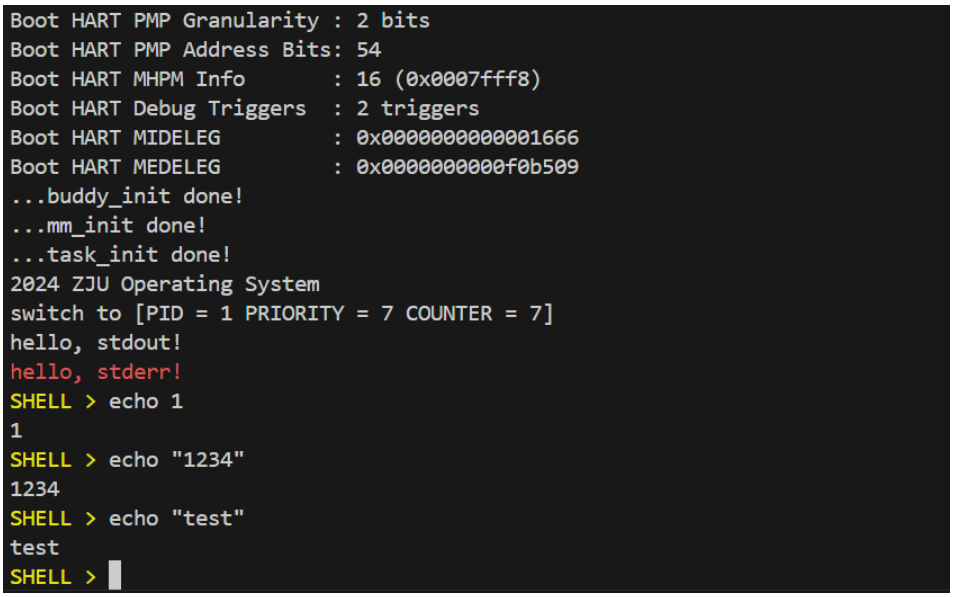

图 1: 实验结果

可以看到写入测试内容正常运行:

```
    write(1, "hello, stdout!\n", 15);
    write(2, "hello, stderr!\n", 15);
```

并且 echo 系统调用实现正确.

5 遇到的问题及解决方法

问题: 编译时找不到引用的函数.

解决方法: 我们需要在根目录的 makefile 下开启对 fs 子目录的编译, 而这点并未在文档中提到

6 心得体会

这次实验让我对于虚拟文件系统有了更深的理解, 它将不同文件抽象成文件描述符, 然后通过打开文件时对文件操作函数指针赋值来实现异构文件的操作