

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

- 1 实验内容及简要原理介绍
- 2 实验具体过程与代码实现
 - 2.1 Shell: 与内核进行交互
 - 2.1.1 文件系统抽象
 - 2.1.2 `stdout/err/in` 初始化
 - 2.1.3 处理 `stdout/err` 的写入
 - 2.1.4 处理 `stdin` 的读取
- 3 实验结果与分析
- 4 心得体会

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_next)
        if(vma->vm_start <= addr && addr < vma->vm_end){
            return vma;
        }
    return NULL;
}
```

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：

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

```
src > lab6 > arch > riscv > M Makefile
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
8
9 clean:
10  ${MAKE} -C kernel clean
11  $(shell test -d boot && rm -rf boot)
12
```

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 > M Makefile
19
20 ## added by lab6
21 LOG := 1
22 ## __added by lab6
23
24 .PHONY:all run debug clean
25
26 all: clean
27   ${MAKE} -C lib all
28   ${MAKE} -C fs all
29   ${MAKE} -C init all
30   ${MAKE} -C user all
31   ${MAKE} -C arch/riscv all
32
33   @echo -e '\n'Build Finished OK
34
35 run: all
36   @echo Launch qemu...
37   @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default
38
39 debug: all
40   @echo Launch qemu for debug...
41   @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s
42
43 clean:
44   ${MAKE} -C lib clean
45   ${MAKE} -C fs clean
46   ${MAKE} -C init clean
47   ${MAKE} -C user clean
48   ${MAKE} -C arch/riscv clean
49   $(shell test -f vmlinux && rm vmlinux)
50   $(shell test -f vmlinux.asm && rm vmlinux.asm)
51   $(shell test -f System.map && rm System.map)
52   @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
1  #ifndef __PROC_H__
60  struct task_struct {
61      uint64_t state;      // 线程状态
62      uint64_t counter;    // 运行剩余时间
63      uint64_t priority;   // 运行优先级 1 最低 10 最高
64      uint64_t pid;       // 线程 id
65
66      struct thread_struct thread;
67      //added in Lab4
68      //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
69      uint64_t *pgd;      // 用户态页表
70      //end added in Lab4
71
72      //added in Lab5
73      struct mm_struct mm;
74      //end added in Lab5
75
76      // added in Lab6
77      struct files_struct *files;
78      // end added in Lab6
79 };
80
81 //added in Lab5
82
```

2.1.2 stdout/err/in 初始化

定义了一个函数 `file_init`:

```
src > lab6 > fs > C fs.c > ...
1  #include "fs.h"
7
8  struct files_struct *file_init() {
9      // todo: alloc pages for files_struct, and initialize stdin, stdout, stderr
10     struct files_struct *ret = (struct files_struct *)alloc_pages(PGROUNDUP(sizeof(struct files_struct)));
11     memset(ret, 0, sizeof(PGROUNDUP(sizeof(struct files_struct))));
12     ret->fd_array[0].opened = 1;
13     ret->fd_array[0].perms = FILE_READABLE;
14     ret->fd_array[0].cfo = 0;
15     ret->fd_array[0].lseek = NULL;
16     ret->fd_array[0].write = NULL;
17     ret->fd_array[0].read = (int64_t*)stdin_read;
18     memcpy(ret->fd_array[0].path, "stdin", 6);
19
20     ret->fd_array[1].opened = 1;
21     ret->fd_array[1].perms = FILE_WRITABLE;
22     ret->fd_array[1].cfo = 0;
23     ret->fd_array[1].lseek = NULL;
24     ret->fd_array[1].write = (int64_t*)stdout_write;
25     ret->fd_array[1].read = NULL;
26     memcpy(ret->fd_array[1].path, "stdout", 7);
27
28
29     ret->fd_array[2].opened = 1;
30     ret->fd_array[2].perms = FILE_WRITABLE;
31     ret->fd_array[2].cfo = 0;
32     ret->fd_array[2].lseek = NULL;
33     ret->fd_array[2].write = (int64_t*)stderr_write;
34     ret->fd_array[2].read = NULL;
35     memcpy(ret->fd_array[2].path, "stderr", 7);
36     return ret;
37 }
38
```

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)
1  #ifndef __STRING_H__
2  #define __STRING_H__
3
4  #include "stdint.h"
5
6  void *memset(void *, int, uint64_t);
7
8  void *memcpy(void *, void *, uint64_t);
9
10 int memcmp(const void *, const void *, uint64_t);
11
12 int strlen(const char *);
13
14 #endif
15

src > lab6 > lib > C string.c > memcmp(const void *, const void *, uint64_t)
1  #include "string.h"
2  #include "stdint.h"
3
4  void *memset(void *dest, int c, uint64_t n) {
5      char *s = (char *)dest;
6      for (uint64_t i = 0; i < n; ++i) {
7          s[i] = c;
8      }
9      return dest;
10 }
11
12 //added in Lab4
13 void *memcpy(void *dest, void *src, uint64_t n) {
14     char *d = dest;
15     char *s = src;
16     while (n-- > 0) {
17         *(d++) = *(s++);
18     }
19     return dest;
20 }
21 //end added in Lab4
22
23 int memcmp(const void *s1, const void *s2, uint64_t n) {
24     const unsigned char *c1 = s1, *c2 = s2;
25     while (n-- > 0) {
26         if (*c1 != *c2) {
27             return *c1 - *c2;
28         }
29         c1++;
30         c2++;
31     }
32     return 0;
33 }
34
35 int strlen(const char *s) {
36     int len = 0;
37     while (s[len]) {
38         len++;
39     }
40     return len;
41 }
```

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
86
87 int64_t my_sys_write(uint64_t fd, const char *buf, uint64_t len) {
88     int64_t ret;
89     struct file *file = &(current->files->fd_array[fd]);
90     if (file->opened == 0) {
91         printk("file not opened\n");
92         return ERROR_FILE_NOT_OPEN;
93     } else {
94         // check perms and call write function of file
95         if (file->perms & FILE_WRITABLE) {
96             ret = file->write(file, buf, len);
97         } else {
98             printk("file not writable\n");
99             ret = -1;
100         }
101     }
102     return ret;
103 }
104
105 int64_t my_sys_read(uint64_t fd, char *buf, uint64_t len) {
106     int64_t ret;
107     struct file *file = &(current->files->fd_array[fd]);
108     if (file->opened == 0) {
109         printk("file not opened\n");
110         return ERROR_FILE_NOT_OPEN;
111     } else {
112         // check perms and call read function of file
113         if (file->perms & FILE_READABLE) {
114             ret = file->read(file, buf, len);
115         } else {
116             printk("file not readable\n");
117             ret = -1;
118         }
119     }
120     return ret;
121 }
122

src > lab6 > arch > riscv > kernel > C syscall.c > do_syscall_pt_regs
123
124 void do_syscall(struct pt_regs *regs) {
125     switch (regs->reg17) {
126     case SYS_WRITE:
127         regs->reg10 = my_sys_write(regs->reg10, (const char *)regs->reg11, regs->reg12);
128         break;
129     case SYS_READ:
130         regs->reg10 = my_sys_read(regs->reg10, (const char *)regs->reg11, regs->reg12);
131         break;
132     case SYS_GETPID:
133         regs->reg10 = current->pid;
134         break;
135     case SYS_CLONE:
136         regs->reg10 = do_fork(regs);
137         break;
138     default:
139         Err("not support syscall id = %d", regs->reg7);
140     }
141     regs->sepc += 4;
142 }
```

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：

```
src > lab6 > arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t, pt_regs *)
6 void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
9     switch (highest_bit)
16     }else{
17         Err("[S] Unhandled Interrupt: scause=%d, sepc=%llx", scause, sepc);
18     }
19     break;
20 case 0:
21     if(scause == 0x8){
22         // printk("Environment call from U-mode\n");
23         // switch (regs->reg17)
24         // {
25         // case SYS_WRITE://sys_write
26         //     sys_write(regs->reg10, (const char*)regs->reg11, (uint64_t)regs->reg12, regs);
27         //     break;
28         // case SYS_GETPID://sys_getpid
29         //     // Log("[PID = %d PC = 0x%x] sys_getpid", current->pid, regs->sepc);
30         //     sys_getpid(regs);
31         //     break;
32         // case SYS_CLONE://sys_clone
33         //     // Log("[PID = %d PC = 0x%x] sys_clone", current->pid, regs->sepc);
34         //     regs->reg10 = do_fork(regs);
35         //     break;
36         // }
37         // regs->sepc += 0x4;
38
39         // added by lab6
40         do_syscall(regs);
41         // end added by lab6
42     }
```

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：

```
src > lab6 > fs > C vfs.c > ...
17
18 int64_t stdin_read(struct file *file, void *buf, uint64_t len) {
19     // todo: use uart_getchar() to get 'len' chars
20     for(int i = 0; i < len; i++){
21         ((char*)buf)[i] = uart_getchar();
22     }
23 }
24
25 int64_t stdout_write(struct file *file, const void *buf, uint64_t len) {
26     char to_print[len + 1];
27     for (int i = 0; i < len; i++) {
28         to_print[i] = ((const char *)buf)[i];
29     }
30     to_print[len] = 0;
31     return printk(buf);
32 }
33
34 int64_t stderr_write(struct file *file, const void *buf, uint64_t len) {
35     char to_print[len + 1];
36     for (int i = 0; i < len; i++) {
37         to_print[i] = ((const char *)buf)[i];
38     }
39     to_print[len] = 0;
40     return printk(buf);
41 }
42
```

现在我们已经可以执行输出了：

```
Boot HART MEDELEG : 0x0000000000001000
Boot HART MEDELEG : 0x000000000000f0b509
...buddy_init done!
...mm_init done!
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
hello, stdout!
hello, stderr!
SHELL >
Select folder. 1
```

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

```
src > lab6 > arch > riscv > include > C sbi.h > sbi_debug_console_read(unsigned long, unsigned long, un
1 #ifndef __SBI_H__
27
28 struct sbiret sbi_debug_console_read(unsigned long num_bytes,
29 unsigned long base_addr_lo, unsigned long base_addr_hi);
30
31 #endif
32

src > lab6 > arch > riscv > kernel > C sbi.c > sbi_debug_console_read(unsigned long, unsigned long, unsigned long)
50
51
52 struct sbiret sbi_debug_console_read(unsigned long num_bytes,
53 unsigned long base_addr_lo, unsigned long base_addr_hi){
54     return sbi_ecall(0x4442434E, 1, num_bytes, base_addr_lo, base_addr_hi, 0, 0, 0);
55 }
```

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 eid, uint64_t fid,
1  #include "stdint.h"
2  #include "sbi.h"
3
4  struct sbiret sbi_ecall(uint64_t eid, uint64_t fid,
5                          uint64_t arg0, uint64_t arg1, uint64_t arg2,
6                          uint64_t arg3, uint64_t arg4, uint64_t arg5) {
7      struct sbiret ret;
8      uint64_t error, value;
9      asm volatile (
10         "mv a7, %[ob_e]\n"
11         "mv a6, %[ob_f]\n"
12         "mv a0, %[ob_0]\n"
13         "mv a1, %[ob_1]\n"
14         "mv a2, %[ob_2]\n"
15         "mv a3, %[ob_3]\n"
16         "mv a4, %[ob_4]\n"
17         "mv a5, %[ob_5]\n"
18         "ecall\n"
19         "mv %[error], a0\n"
20         "mv %[value], a1\n"
21         : [error] "=r" (error), [value] "=r" (value)
22         : [ob_e] "r" (eid), [ob_f] "r" (fid),
23           [ob_0] "r" (arg0), [ob_1] "r" (arg1),
24           [ob_2] "r" (arg2), [ob_3] "r" (arg3),
25           [ob_4] "r" (arg4), [ob_5] "r" (arg5)
26         : "memory", "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
27     );
28     ret.error = error;
29     ret.value = value;
30     return ret;
31 }

```

```

src > lab6 > user > C printf.c > printf(const char *, ...)
287 int printf(const char* s, ...) {
288     va_list vl;
289     va_start(vl, s);
290     res = vprintfmt(putc, s, vl);
291     long syscall_ret, fd = 1;
292     buffer[tail++] = '\0';
293     asm volatile ("li a7, %1\n"
294                  "mv a0, %2\n"
295                  "mv a1, %3\n"
296                  "mv a2, %4\n"
297                  "ecall\n"
298                  "mv %0, a0\n"
299                  : "+r" (syscall_ret)
300                  : "i" (SYS_WRITE), "r" (fd), "r" (&buffer), "r" (tail)
301                  : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7");
302     tail = 0;
303     va_end(vl);
304     return res;
305 }
306
307

```

3 实验结果与分析

贴出 `echo` 命令

```
...buddy_init done!  
...mm_init done!  
...task_init done!  
2024 ZJU Operating System  
SET [PID = 1 PRIORITY = 7 COUNTER = 7]  
  
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]  
hello, stdout!  
hello, stderr!  
SHELL > echo "test"  
test  
SHELL > echo "test"  
test  
SHELL > 
```

4 心得体会

Nice OS!