Lab7: VFS & FAT32 文件系统

1 准备工作

同步实验框架，不赘述。

在`fs/virtio.c`添加`virt_to_phys`定义

```
// fs/virtio.c
uint64_t virt_to_phys(uint64_t virt){
    return virt - PA2VA_OFFSET;
}
```

删除`uapp.S`和`getpid.c`，修改MakeFile。

此外，向 `include/types.h` 中补充一些类型别名。

```
typedef unsigned long uint64_t;
typedef long int64_t;
typedef unsigned int uint32_t;
typedef int int32_t;
typedef unsigned short uint16_t;
typedef short int16_t;
typedef uint64_t* pagetable_t;
typedef char int8_t;
typedef unsigned char uint8_t;
typedef uint64_t size_t;
```

还要修改一下 `arch/riscv/kernel/vmlinux.lds` 中的 `_sramdisk` 符号部分(将 uapp 修改为 ramdisk)

2 处理 `stdout` 的写入

修改`syscall.c`来处理read和write的syscall

```
void syscall(struct pt_regs* regs) {
    if (regs->x[17] == SYS_WRITE) {
        regs->x[10] = sys_write((unsigned int)regs->x[10], (const char*)regs->x[11], regs-
>x[12]);
    }
    else if (regs->x[17] == SYS_GETPID) {
        regs->x[10] = current->pid;
    }
    else if (regs->x[17] == SYS_READ){
        regs->x[10] = sys_read((unsigned int)regs->x[10], (char*)regs->x[11], regs-
>x[12]);
    }
    else {
        printk("Unhandled Syscall: 0x%lx\n", regs->x[17]);
        while (1);
```

```
    }
    regs->sepc += 4;
}
```

在创建进程时为进程初始化文件，当初始化进程时，先完成打开的文件的列表的初始化，这里我们的方式是直接分配一个页，并用 `files` 指向这个页。

```
struct file* file_init() {
    struct file *ret = (struct file*)alloc_page();

    // stdin
    ret[0].opened = 1;
    ret[0].perms = FILE_READABLE;
    ret[0].cfo = 0;
    ret[0].lseek = NULL;
    ret[0].write = NULL;
    ret[0].read = stdin_read;
    memcpy(ret[0].path, "stdin", 6);

    // stdout
    ret[1].opened = 1;
    ret[1].perms = FILE_WRITABLE;
    ret[1].cfo = 0;
    ret[1].lseek = NULL;
    ret[1].write = stdout_write/* todo */;
    ret[1].read = NULL;
    memcpy(ret[1].path, "stdout", 7);

    // stderr
    ret[2].opened = 1;
    ret[2].perms = FILE_WRITABLE;
    ret[2].cfo = 0;
    ret[2].lseek = NULL;
    ret[2].write = stderr_write;
    ret[2].read = NULL;
    memcpy(ret[2].path, "stdout", 7);

    return ret;
}
```

在`proc.c`的进程初始化时调用`file_init()`

```
// proc.c
task[i]->files = file_init();
```

实现`sys_write`

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

3 处理 `stderr` 的写入

仿照 `stdout` 的输出过程，完成 `stderr` 的写入，让 `nish` 可以正确打印出

```
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(buf);
}
```

4 处理 `stdin` 的读取

代码框架中已经实现了一个在内核态用于向终端读取一个字符的函数，要调用这个函数来实现自己的 `stdin_read`

```
char uart_getchar() {
    char ret;
    while (1) {
        struct sbiret sbi_result = sbi_ecall(SBI_GETCHAR, 0, 0, 0, 0, 0, 0, 0);
        if (sbi_result.error != -1) {
            ret = sbi_result.error;
            break;
        }
    }
    return ret;
}

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;
}
```

接着参考 `syscall_write` 的实现，来实现 `syscall_read`

```
int64_t sys_read(unsigned int fd, char* buf, uint64_t count) {
    int64_t ret;
    struct file* target_file = &(current->files[fd]);
    if (target_file->opened) {
        ret = target_file->read(target_file, (const void*)buf, count);
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}
```

5 实验结果

```
2022 Hello RISC-V
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
hello, stdout!
hello, stderr!
SHELL > echo "this is echo"
this is echo
SHELL >
```