Lab7: VFS & FAT32 文件系统

本实验中不涉及 `fork` 的实现和缺页异常，只需要完成 Lab4 即可开始本实验。

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）。
- 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）。

实验环境

与先前的实验中使用的环境相同。

背景知识

VFS

虚拟文件系统(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、经典Mac OS/macOS和Unix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。

VirtIO

Virtio 是一个开放标准，它定义了一种协议，用于不同类型的驱动程序和设备之间的通信。在 QEMU 上我们可以基于 VirtIO 使用许多模拟出来的外部设备，在本次实验中我们使用 VirtIO 模拟存储设备，并在其上构建文件系统。

MBR

主引导记录（英语：Master Boot Record，缩写：MBR），又叫做主引导扇区，是计算机开机后访问硬盘时所必须要读取的首个扇区，它在硬盘上位于第一个扇区。在深入讨论主引导扇区内部结构的时候，有时也将其开头的446字节内容特指为"主引导记录"（MBR），其后是4个16字节的"磁盘分区表"（DPT），以及2字节的结束标志（55AA）。因此，在使用"主引导记录"（MBR）这个术语的时候，需要根据具体情况判断其到底是指整个主引导扇区，还是主引导扇区的前446字节。

FAT32

文件分配表（File Allocation Table，首字母缩略字：FAT），是一种由微软发明并拥有部分专利的文件系统，供MS-DOS使用，也是所有非NT核心的Windows系统使用的文件系统。最早的 FAT 文件系统直接使用扇区号来作为存储的索引，但是这样做的缺点是显然易见的：当磁盘的大小不断扩大，存储扇区号的位数越来越多，越发占用存储空间。以 32 位扇区号的文件系统为例，如果磁盘的扇区大小为 512B，那么文件系统能支持的最大磁盘大小仅为 2TB。

所以在 FAT32 中引入的新的存储管理单位“簇”，一个簇包含一个或多个扇区，文件系统中记录的索引不再为扇区号，而是**簇号**，以此来支持更大的存储设备。你可以参考这些资料来学习 FAT32 文件系统的标准与实现：

- FAT32文件系统？盘它！
- Microsoft FAT Specification

实验步骤

Shell: 与内核进行交互

我们为大家提供了 `nish` 来与我们在实验中完成的 kernel 进行交互。`nish` (Not Implemented SHell) 提供了简单的用户交互和文件读写功能，有如下的命令。

```
echo [string] # 将 string 输出到 stdout
cat  [path]   # 将路径为 path 的文件的内容输出到 stdout
edit [path] [offset] [string] # 将路径为 path 的文件，
            # 偏移量为 offset 的部分开始，写为 string
```

同步 `os23fall-stu` 中的 `user` 文件夹，替换原有的用户态程序为 `nish`。为了能够正确启动 QEMU，需要下载磁盘镜像并放置在项目目录下。

```
lab7
├── Makefile
├── disk.img
├── arch
│   └── riscv
│       ├── Makefile
│       └── include
│           └── sbi.h
├── fs
│   ├── Makefile
│   ├── fat32.c
│   ├── fs.S
│   ├── mbr.c
│   ├── vfs.c
│   └── virtio.c
├── include
│   ├── fat32.h
│   ├── fs.h
│   ├── mbr.h
│   ├── string.h
│   ├── debug.h
│   └── virtio.h
├── lib
│   └── string.c
└── user
    ├── Makefile
    ├── forktest.c
    ├── link.lds
    ├── printf.c
    ├── ramdisk.S
    ├── shell.c
    ├── start.S
```

```
    ├── stddef.h
    ├── stdio.h
    ├── string.h
    ├── syscall.h
    ├── unistd.c
    └── unistd.h
```

此外，可能还要向 `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;
```

完成这一步后，可能你还需要调整一部分头文件引用和 `Makefile`，以让项目能够成功编译并运行。

我们在启动一个用户态程序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`。在 `nish` 启动时，会首先向 `stdout` 和 `stdout` 分别写入一段内容，用户态的代码如下所示。

```
// user/shell.c

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

处理 `stdout` 的写入

我们在用户态已经像上面这样实现好了 `write` 函数来向内核发起 syscall，我们先在内核态完成真实的写入过程，也即将写入的字符输出到串口。

```
// arch/riscv/include/syscall.h

int64_t sys_write(unsigned int fd, const char* buf, uint64_t count);

// arch/riscv/include/syscall.c

void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs) {
    ...
    if (scause == 0x8) { // syscalls
        uint64_t sys_call_num = regs->a7;
        ...
        if (sys_call_num == SYS_WRITE) {
            regs->a0 = sys_write(regs->a0, (const char*)(regs->a1), regs->a2);
            regs->sepc = regs->sepc + 4;
        } else {
            printk("Unhandled Syscall: 0x%lx\n", regs->a7);
            while (1);
        }
```

```
    }
    ...
}
```

注意到我们使用的是 `fd` 来索引打开的文件，所以在该进程的内核态需要维护当前进程打开的文件，将这些文件的信息储存在一个表中，并在 `task_struct` 中指向这个表。

```
// include/fs.h

struct file {
    uint32_t opened;
    uint32_t perms;
    int64_t cfo;
    uint32_t fs_type;

    union {
        struct fat32_file fat32_file;
    };

    int64_t (*lseek) (struct file* file, int64_t offset, uint64_t whence);
    int64_t (*write) (struct file* file, const void* buf, uint64_t len);
    int64_t (*read)  (struct file* file, void* buf, uint64_t len);

    char path[MAX_PATH_LENGTH];
};

// arch/riscv/include/proc.h

struct task_struct {
    ...
    struct file *files;
    ...
};
```

首先要做的是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。

```
// fs/vfs.c
int64_t stdout_write(struct file* file, const void* buf, uint64_t len);
int64_t stderr_write(struct file* file, const void* buf, uint64_t len);
int64_t stdin_read(struct file* file, void* buf, uint64_t len);

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;
    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, "stderr", 7);

    return ret;
}

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(buf);
}

// arch/riscv/kernel/proc.c
void task_init() {
    ...
    // Initialize the stdin, stdout, and stderr.
    task[1]->files = file_init();
    printk("[S] proc_init done!\n");
    ...
}
```

可以看到每一个被打开的文件对应三个函数指针，这三个函数指针抽象出了每个被打开的文件的操作。也对应了 `SYS_LSEEK`，`SYS_WRITE`，和 `SYS_READ` 这三种 syscall. 最终由函数 `sys_write` 调用 `stdout` 对应的 `struct file` 中的函数指针 `write` 来执行对应的写串口操作。我们这里直接给出 `stdout_write` 的实现，只需要直接把这个函数指针赋值给 `stdout` 对应 `struct file` 中的 `write` 即可。

接着你需要实现 `sys_write` syscall，来间接调用我们赋值的 `stdout` 对应的函数指针。

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

```
            ret = ERROR_FILE_NOT_OPEN;
        }
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}
```

至此，你已经能够打印出 `stdout` 的输出了。

```
2023 Hello RISC-V
hello, stdout!
```

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

```
2023 Hello RISC-V
hello, stdout!
hello, stderr!
SHELL >
```

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。

在终端中已经实现了不断读 `stdin` 文件来获取键入的内容，并解析出命令，你需要完成的只是响应如下的系统调用：

```
// user/shell.c

read(0, read_buf, 1);
```

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，你需要调用这个函数来实现你的 `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) {
    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 = &(get_current_task()->files[fd]);
    if (target_file->opened) {
        if (target_file->perms & FILE_READABLE) {
            ret = target_file->read(target_file, buf, count);
        } else {
            printk("file not readable\n");
            ret = ERROR_FILE_NOT_OPEN;
        }
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}
```

至此，就可以在 `nish` 中使用 `echo` 命令了。

```
SHELL > echo "this is echo"
this is echo
```