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。

1. 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）。

2. 实验环境

与先前的实验中使用的环境相同。

- https://academy.cba.mit.edu/classes/networking_communications/SD/FAT.pdf)

3. 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 `stderr` 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。

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。

实现的`file_init`如下：

```
// fs/vfs.c
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 = (int64_t *)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/* todo */;
    ret[2].read = NULL;
    memcpy(ret[2].path, "stderr", 7);

    return ret;
}
```

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。

```

// arch/riscv/kernel/syscall.c
uint64_t sys_write(unsigned int fd, const char* buf, uint64_t count) {
    uint64_t ret;
    struct file* target_file = &(current->files[fd]);
    if (target_file->opened) {
        target_file->write(target_file, buf, count);
    } else {
        printk("file not open_write\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`，具体实现如下.

```
// fs/vfs.c
char uart_getchar() {
    /* already implemented in the file */
}

int64_t stdin_read(struct file* file, void* buf, uint64_t len) {
    char *res = (char *) buf;
    for(int i = 0; i < len; i++){
       res[i] = uart_getchar();
    }
    return len;
}
```

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

```
// arch/riscv/kernel/syscall.c
uint64_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) {
         target_file->read(target_file, buf, count);
    } else {
        printk("file not open_read\n");
        ret = ERROR_FILE_NOT_OPEN;
        while(1);
    }
    return ret;
}
```

至此，就可以在 `nish` 中使用 `echo` 命令了。

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

- 由于时间原因，只实现了lab7的第一部分，第二部分没有实现

FAT32: 持久存储

在本次实验中我们仅需实现 FAT32 文件系统中很小一部分功能，我们为实验中的测试做如下限制：

- 文件名长度小于等于 8 个字符，并且不包含后缀名和字符 `.`.
- 不包含目录的实现，所有文件都保存在磁盘根目录下。
- 不涉及磁盘上文件的创建和删除。
- 不涉及文件大小的修改。

准备工作

利用 VirtIO 为 QEMU 添加虚拟存储

我们为大家构建好了磁盘镜像，其中包含了一个 MBR 分区表以及一个存储有一些文件的 FAT32 分区。可以使用如下的命令来启动 QEMU，并将该磁盘连接到 QEMU 的一个 VirtIO 接口上，构成一个 `virtio-blk-device`。

```
run: all
    @echo Launch the qemu ......
    @qemu-system-riscv64 \
        -machine virt \
        -nographic \
        -bios default \
        -kernel vmlinux \
        -global virtio-mmio.force-legacy=false \
        -drive file=disk.img,if=none,format=raw,id=hd0 \
        -device virtio-blk-device,drive=hd0
```

`virtio` 所需的驱动我们已经为大家编写完成了，在 `fs/virtio.c` 中给出。

然后在创建虚拟内存映射时，还需要添加映射 VritIO 外设部分的映射。

```
// arch/riscv/kernel/vm.c
create_mapping(task->pgd, io_to_virt(VIRTIO_START), VIRTIO_START, VIRTIO_SIZE
* VIRTIO_COUNT, PTE_W | PTE_R | PTE_V);
```

初始化 MBR

我们为大家实现了读取 MBR 这一磁盘初始化过程。该过程会搜索磁盘中存在的分区，然后对分区进行初步的初始化。

对 VirtIO 和 MBR 进行初始化的逻辑可以被添加在初始化第一个进程的 `task_init` 中

```
// arch/riscv/kernel/proc.c
void task_init() {
    ...
    printk("[S] proc_init done!\n");

    virtio_dev_init();
    mbr_init();
}
```

这样从第一个用户态进程被初始化完成开始，就能够直接使用 VirtIO，并使用初始化完成的 MBR 表了。

初始化 FAT32 分区

在 FAT32 分区的第一个扇区中存储了关于这个分区的元数据，首先需要读取并解析这些元数据。我们提供了两个数据结构的定义，`fat32_bpb` 为 FAT32 BIOS Parameter Block 的简写。这是一个物理扇区，其中对应的是这个分区的元数据。首先需要将该扇区的内容读到一个 `fat32_bpb` 数据结构中进行解析。`fat32_volume` 是用来存储我们实验中需要用到的元数据的，需要根据 `fat32_bpb` 中的数据来进行计算并初始化。

```
// fs/fat32.c
```

```
2
3  struct fat32_bpb fat32_header;      // FAT32 metadata in the disk
4  struct fat32_volume fat32_volume;   // FAT32 metadata to initialize
5
6  void fat32_init(uint64_t lba, uint64_t size) {
7      virtio_blk_read_sector(lba, (void*)&fat32_header);
8      fat32_volume.first_fat_sec = /* to calculate */;
9      fat32_volume.sec_per_cluster = /* to calculate */;
10     fat32_volume.first_data_sec = /* to calculate */;
11     fat32_volume.fat_sz = /* to calculate */;
12
13     virtio_blk_read_sector(fat32_volume.first_data_sec, fat32_buf); // Get
   the root directory
14     struct fat32_dir_entry *dir_entry = (struct fat32_dir_entry *)fat32_buf;
15 }
16
```

读取 FAT32 文件

在读取文件之前，首先需要打开对应的文件，这需要实现 `openat` syscall.

```
1  // arch/riscv/syscall.c
2
3  int64_t sys_openat(int dfd, const char* filename, int flags) {
4      int fd = -1;
5
6      // Find an available file descriptor first
7      for (int i = 0; i < PGSIZE / sizeof(struct file); i++) {
8          if (!current->files[i].opened) {
9              fd = i;
10             break;
11         }
12     }
13
14     // Do actual open
15     file_open(&(current->files[fd]), filename, flags);
16
17     return fd;
18 }
19
20 void file_open(struct file* file, const char* path, int flags) {
21     file->opened = 1;
22     file->perms = flags;
23     file->cfo = 0;
24     file->fs_type = get_fs_type(path);
25     memcpy(file->path, path, strlen(path) + 1);
26
27     if (file->fs_type == FS_TYPE_FAT32) {
28         file->lseek = fat32_lseek;
29         file->write = fat32_write;
30         file->read = fat32_read;
31         file->fat32_file = fat32_open_file(path);
32     } else if (file->fs_type == FS_TYPE_EXT2) {
33         printk("Unsupport ext2\n");
34         while (1);
```

```
    } else {
        printk("Unknown fs type: %s\n", path);
        while (1);
    }
}
```

我们使用最简单的判别文件系统的方式，文件前缀为 `/fat32/` 的即是本次 FAT32 文件系统中的文件，例如，在 `nish` 中我们尝试读取文件，使用的命令是 `cat /fat32/$FILENAME`. `file_open` 会根据前缀决定是否调用 `fat32_open_file` 函数。注意因为我们的文件一定在根目录下，也即 `/fat32/` 下，无需实现与目录遍历相关的逻辑。此外需要注意的是，需要将文件名统一转换为大写或小写，因为我们的实现是不区分大小写的。

```
// arch/riscv/syscall.c

struct fat32_file fat32_open_file(const char *path) {
    struct fat32_file file;
    /* todo: open the file according to path */
    return file;
}
```

在打开文件后自然是进行文件的读取操作，需要先实现 `lseek` syscall. 注意实现之后需要在打开文件时将对应的 `fat32_lseek` 赋值到打开的 FAT32 文件系统中的文件的 `lseek` 函数指针上。

```
// arch/riscv/kernel/syscall.c

int64_t sys_lseek(int fd, int64_t offset, int whence) {
    int64_t ret;
    struct file* target_file = &(current->files[fd]);
    if (target_file->opened) {
        /* todo: indirect call */
    } else {
        printk("file not open\n");
        ret = ERROR_FILE_NOT_OPEN;
    }
    return ret;
}

// fs/fat32.c

int64_t fat32_lseek(struct file* file, int64_t offset, uint64_t whence) {
    if (whence == SEEK_SET) {
        file->cfo = /* to calculate */;
    } else if (whence == SEEK_CUR) {
        file->cfo = /* to calculate */;
    } else if (whence == SEEK_END) {
        /* Calculate file length */
        file->cfo = /* to calculate */;
    } else {
        printk("fat32_lseek: whence not implemented\n");
        while (1);
    }
    return file->cfo;
}
```

然后需要完成 `fat32_read` 并将其赋值给打开的 FAT32 文件的 `read` 函数指针。

```
// fs/fat32.c

int64_t fat32_read(struct file* file, void* buf, uint64_t len) {
    /* todo: read content to buf, and return read length */
}
```

完成 FAT32 读的部分后，就已经可以在 `nish` 中使用 `cat /fat32/email` 来读取到在磁盘中预先存储的一个名为 email 的文件了。

当然，最后还需要完成 `close` syscall 来将文件关闭。

写入 FAT32 文件

在完成读取后，就可以仿照读取的函数完成对文件的修改。在测试时可以使用 `edit` 命令在 `nish` 中对文件做出修改。需要实现 `fat32_write`，可以参考前面的 `fat32_read` 来进行实现。

```
// fs/fat32.c

int64_t fat32_write(struct file* file, const void* buf, uint64_t len) {
    /* todo: fat32_write */
}
```

测试

这里只完成了第一部分，不涉及第二部分的测试

```
Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
...set_vm_final successfully!
...proc_init done!
[S-MODE] Hello RISC-V
hello, stdout!
hello, stderr!
SHELL > echo "this is a file"
this is a file
SHELL >
```