Lab5实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

thinking 5.1

kseg0 是存放内核的区域，一般通过 cache 访问。如果在写设备的时候将写入缓存到cache，因为cache只有在置换时才会写回，因此后面的操作会覆盖前面的操作，只有最后被换出时才会写回。

对于不同的设备，例如，串口设备相对于IDE磁盘读写更频繁，因此相同时间内产生错误的概率更大

thinking 5.2

在user/include/fs.h中，有如下定义

```
#define BY2BLK BY2PG
#define BY2FILE 256
#define MAXPATHLEN 1024

struct File {
    char f_name[MAXNAMELEN]; // filename
    uint32_t f_size;     // file size in bytes
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT];
    uint32_t f_indirect;

    struct File *f_dir; // the pointer to the dir where this file is in, valid
only in memory.
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

在之前的实验中，我们知道BY2PG的大小是4096，所以一个磁盘块大小BY2BLK=4096B=4KB，而一个文件控制块大小BY2FILE=256B，所以一个磁盘块最多存储4KB/256B=16个文件控制块

一个目录最多指向1024个磁盘块，一个磁盘块最多存储16个文件控制块，所以一个目录下最多可以有1024*16=16384个文件

f_indirect 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针，每个指针的大小是 4B，一个磁盘块大小4096B，所以可以存储 1024 个指针。其中前 10 个指针不使用，但是另有 f_direct[NDIRECT] 10个直接指针指向 10 个磁盘块，所以单个文件最多由 1024 个磁盘块构成，大小最大是1024*4096=1K*4K=4MB。

thinking 5.3

在fs/serv.h中

```
/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000
```

我们实验使用的内核支持的最大磁盘大小是1GB

thinking 5.4

- fs/serv.h

```
#define BY2SECT 512         //定义了一个扇区的大小
#define SECT2BLK (BY2BLK / BY2SECT) //定义了一个磁盘块包含的扇区数

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP 0x10000000 //缓存磁盘块的起始地址

/* Maximum disk size we can handle (1GB) */
#define DISKMAX 0x40000000 //我们实验使用的内核支持的最大磁盘大小
```

- user/include/fs.h

```
// Bytes per file system block - same as page size
#define BY2BLK BY2PG //一个磁盘块的大小就是一个页面的大小
#define BIT2BLK (BY2BLK * 8)

// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN 128 //用于存文件名的 char 数组大小

// Maximum size of a complete pathname, including null
#define MAXPATHLEN 1024 //用于存路径的数组大小

// Number of (direct) block pointers in a File descriptor
#define NDIRECT 10 //指向磁盘块的直接指针数
#define NINDIRECT (BY2BLK / 4) //间接引用块的指针数

#define MAXFILESIZE (NINDIRECT * BY2BLK) //单个文件最大大小

#define BY2FILE 256 //一个文件控制块大小
```

thinking 5.5

```
int main() {
    int fd = open("./test", O_RDWR);
    if(-1 == fd) {
        printf("file openerror\n");
        return -1;
    }

    pid_t pid;

    if((pid = fork()) < 0) {
        printf("fork error");
    }else if(pid == 0) {
        struct Fd *tmp = num2fd(fd);
    printk("%d %d", r, tmp->fd_offset);
    }else {
        struct Fd *tmp = num2fd(fd);
    printk("%d %d", r, tmp->fd_offset);
   }
```

```
    return 0;
}
```

结果：父进程和子进程的输出是相同的，这说明 fork 前后的父子进程会共享文件描述符和定位指针

thinking 5.6

```
struct File {
    char f_name[MAXNAMELEN]; // 文件名
    uint32_t f_size; // 文件大小
    uint32_t f_type; // 文件类型
    uint32_t f_direct[NDIRECT];//直接引用指针
    uint32_t f_indirect;//指向间接引用块

    struct File *f_dir; // 指向文件所属的文件目录
    char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));// 让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节

struct Fd {
    u_int fd_dev_id; // 指外设id，也就是外设类型。
    u_int fd_offset; // 读写的当前位置（偏移量），类似“流”的当前位置
    u_int fd_omode; // 指文件打开方式，如只读，只写等
};

struct Filefd {
    struct Fd f_fd; // 文件描述符
    u_int f_fileid; // 文件id
    struct File f_file; //文件本身
};
```

thinking 5.7

内核开始运行时，就启动了文件系统服务进程。

在用户进程中，通过 user/file.c 中的函数操作文件系统，这些函数又调用了 user/fsipc.c 中的函数，而 user/fsipc.c 中的函数又是依靠ipc通信与文件系统进行了通信

在文件系统进程中，首先完成serv_init和fs_init的初始化，然后也是借助ipc通信对用户操作结果进行反馈

二、实验难点

设备驱动

在实现文件系统之前要实现设备驱动

需要实现的设备驱动主要有三种：即 console，IDE_disk 和 rtc。在内存中的布局如下表所示。

device	start addr	length
console	0x10000000	0x20
IDE_disk	0x13000000	0x4200

device	start addr	length
rtc	0x15000000	0x200

其中：

- console 为控制台终端，即我们在编写程序时用于输入输出的地方
- IDE_disk，用于存储文件等
- rtc 为实时时钟终端，用于获取当前时间等信息

在课下实验中，我们实现了对IDE磁盘的读写，在课上第一次实验，我们实现了对rtc实时时钟的读写

首先，我们需要在内核态完成系统对这些设备的读写

```
int sys_write_dev(u_int va, u_int pa, u_int len) {
    /* Exercise 5.1: Your code here. (1/2) */
    if(is_illegal_va_range(va, len)){
        return -E_INVAL;
    }
    if((pa >= 0x10000000 && pa+len <= 0x10000020) || (pa >= 0x13000000 && pa+len
<= 0x13004200) || (pa >= 0x15000000 && pa+len <= 0x15000200)){
        memcpy(0xa0000000+pa, va, len);
        return 0;
    }
    return -E_INVAL;
}
int sys_read_dev(u_int va, u_int pa, u_int len) {
    /* Exercise 5.1: Your code here. (2/2) */
    if(is_illegal_va_range(va, len)){
        return -E_INVAL;
    }
    if((pa >= 0x10000000 && pa+len <= 0x10000020) || (pa >= 0x13000000 && pa+len
<= 0x13004200) || (pa >= 0x15000000 && pa+len <= 0x15000200)){
        memcpy(va, 0xa0000000+pa, len);
        return 0;
    }
    return -E_INVAL;
}
```

接下来，就要对设备分别完成系统调用函数，使得能够在用户态进行设备读写

例如对IDE磁盘的读写，ide_write和ide_read两个函数

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs) {
    u_int begin = secno * BY2SECT;
    u_int end = begin + nsecs * BY2SECT;
    u_int zero = 0;
    int r;
    for (u_int off = 0; begin + off < end; off += BY2SECT) {
        uint32_t temp = diskno;
        u_int cur = off + begin;
        /* Exercise 5.3: Your code here. (1/2) */
        if((r = syscall_write_dev((u_int) & temp, DEV_DISK_ADDRESS+DEV_DISK_ID,
4)) < 0){
            user_panic("error");
```

```
        }
        if((r = syscall_write_dev((u_int) & cur, DEV_DISK_ADDRESS, 4)) < 0){
            user_panic("error");
        }
        if((r = syscall_write_dev((u_int) & zero,
DEV_DISK_ADDRESS+DEV_DISK_START_OPERATION, 4)) < 0){
            user_panic("error");
        }
        u_int v;
        syscall_read_dev((u_int) & v, DEV_DISK_ADDRESS+DEV_DISK_STATUS, 4);
        if(!v){
            user_panic("error");
        }
        if((syscall_read_dev((u_int)(dst+off), DEV_DISK_ADDRESS+DEV_DISK_BUFFER,
BY2SECT)) < 0){
            user_panic("error");
        }
    }
}
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs) {
    u_int begin = secno * BY2SECT;
    u_int end = begin + nsecs * BY2SECT;
    u_int one = 1;
    for (u_int off = 0; begin + off < end; off += BY2SECT) {
        uint32_t temp = diskno;
        u_int cur = off + begin;
        /* Exercise 5.3: Your code here. (2/2) */
        if((syscall_write_dev((u_int)(src+off), DEV_DISK_ADDRESS+DEV_DISK_BUFFER,
BY2SECT)) < 0){
            user_panic("error");
        }
        if((syscall_write_dev((u_int) & temp, DEV_DISK_ADDRESS+DEV_DISK_ID, 4)) <
0){
            user_panic("error");
        }
        if((syscall_write_dev((u_int) & cur, DEV_DISK_ADDRESS, 4)) < 0){
            user_panic("error");
        }
        if((syscall_write_dev((u_int) & one,
DEV_DISK_ADDRESS+DEV_DISK_START_OPERATION, 4)) < 0){
            user_panic("error");
        }
        u_int v;
        if((syscall_read_dev((u_int) & v, DEV_DISK_ADDRESS+DEV_DISK_STATUS, 4)) <
0){
            user_panic("error");
        }
    }
}
```

三、实验感想

细节决定成败，这句话在本次实验真的切实感受到了

课下实验中，在内核态编写设备的读写函数时，因为粗心大意，判断设备物理地址有效性时，没有包括起始地址（判断使用>而不是>=），课下的测试没有测试出错，笔者就没有意识到有错误。在课上第一次实验时，rtc时钟的启动需要读/写起始地址，而因为我的错误将起始地址判断为无效，尝试多次都无法成功启动rtc时钟，白白浪费了大量的实验时间去寻找错误。如果课下编写代码时，能更仔细些，就可以避免这样的低级错误，也就不会在课上紧张的测试时感到手足无措