OS —— Lab5实验报告

22373407 王飞阳

一、思考题

Thinking 5.1

使用 kseg0 访问设备会导致缓存一致性问题和数据完整性问题。

缓存机制是为了提高效率而设计的，具体体现为数据发生改变时并不立即写入内存，而是在cache发生替换时才写入。而我们的控制台需要实时的交互，因此若写入kseg0部分，数据会经过cache，甚至可能很久都不被真正写入，因此我们看到的输出可能并不是该部分实际的内容，从而很可能给出错误的交互行为。

不同种类的设备操作有差异。串口设备通常用于通信，实时性要求较高，由于数据写入被缓存，实际传输的数据可能不及时，或者顺序出现问题。串口缓冲区状态变化无法及时反映，导致错误的控制信号处理。

IDE 磁盘用于数据存储，数据一致性和完整性尤为重要。写回缓存可能导致数据没有及时写入磁盘，从而在断电或系统崩溃时导致数据丢失。磁盘上数据更新后，缓存中的旧数据可能导致读取到不一致的数据。

Thinking 5.2

查找有关文件的代码如下：

```
#define BLOCK_SIZE PAGE_SIZE
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)
#define MAXNAMELEN 128
#define MAXPATHLEN 1024
#define NDIRECT 10
#define NINDIRECT (BLOCK_SIZE / 4)
#define MAXFILESIZE (NINDIRECT * BLOCK_SIZE)
#define FILE_STRUCT_SIZE 256
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[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 -
sizeof(void *)];
} __attribute__((aligned(4), packed));
#define FILE2BLK (BLOCK_SIZE / sizeof(struct File))
```

- 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？

  根据上面的节选，sizeof(struct File) = FILE_STRUCT_SIZE = 256B，BLOCK_SIZE = PAGE_SIZE = 4096B，则一个磁盘块最多存储 4096B/256B=16 个文件控制块。

- 一个目录下最多能有多少个文件？

一个目录（File 结构体）最多指向 1024 个磁盘块，一个磁盘块中有 16 个文件控制块，一个目录下，最多有 1024×16=16384 个文件。

- 我们的文件系统支持的单个文件最大为多大?

  File 结构体中，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，所以一个磁盘块可以存储 1024 个指针。其中前 10 个指针没有使用，但是另有 f_direct[NDIRECT] 指向 10 个磁盘块。所以单个文件最多由 1024 个磁盘块构成，大小是 1024×4096=4MB。

Thinking 5.3

查看源代码可知：

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

内核支持的最大磁盘大小是1GB。

Thinking 5.4

`fs/serve.h`

```
#define PTE_DIRTY 0x0004 // file system block cache is dirty

#define SECT_SIZE 512              /* Bytes per disk sector */

#define SECT2BLK (BLOCK_SIZE / SECT_SIZE) /* sectors to a block */

#define DISKMAP 0x10000000  //磁盘块缓冲区起始地址

#define DISKMAX 0x40000000  //最大磁盘大小
```

`user/include/fs.h`

```
#define BLOCK_SIZE PAGE_SIZE  //一个block对应的字节，也就是说，一个block正好等于一页大小
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)  //一个block对应的位数，一字节等于 8 位，所以要乘 8

#define MAXNAMELEN 128  //用于存文件名的 char 数组大小，由于最后一个必定为'\0',所以只能存 127 个字符
#define MAXPATHLEN 1024  //和上面的类似，只不过是用来存路径的

#define NDIRECT 10  // 直接引用的个数，可以认为是 10 个指针，这里用 int 存 block 的下标来代替了指针的作用
#define NINDIRECT (BLOCK_SIZE / 4)  // 间接引用块的指针个数，由于一个int是 32 位，也就是 4byte，所以是除 4

#define MAXFILESIZE (NINDIRECT * BLOCK_SIZE)  //// 最大文件大小，那么就是引用指针的个数乘以一块的大小

#define FILE_STRUCT_SIZE 256  // 定义了一个 File 结构体（文件索引结构体）所占用的大小

#define FILE2BLK (BLOCK_SIZE / sizeof(struct File))  // 就是一个 block 能容纳多少个 file 索引
```

```
#define FTYPE_REG 0 // Regular file  // 普通文件
#define FTYPE_DIR 1 // Directory  // 目录

#define FS_MAGIC 0x68286097 // Everyone's favorite OS class
```

Thinking 5.5

fork 前后的父子进程会共享文件描述符和定位指针。

测试代码：

```
int fdnum = open("/newmotd", O_RDWR);
if (fdnum < 0) {
    user_panic("open failed");
}
seek(fdnum, 114514);
int pid = fork();
if (pid == 0) {
    struct Fd *fd;
    if (fd_lookup(fdnum, &fd) < 0) {
        debugf("No!\n");
    } else {
        debugf("Yes, THE CHILD: fd->fd_offset = %d\n", fd->fd_offset);
    }
}
```

运行结果：

```
init.c: mips_init() is called
Memory size: 65536 KiB, number of pages: 16384
to memory 80430000 for struct Pages.
pmap.c:  mips vm init success
FS is running
superblock is good
read_bitmap is good
open is good
Yes, THE CHILD: fd->fd_offset = 114514
```

Thinking 5.6

```
// file descriptor
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。
};

// file descriptor + file
struct Filefd {
    struct Fd f_fd;  //即文件描述符。
    u_int f_fileid;  // 指文件本身的id。
    struct File f_file;  //指文件本身。
};
```

```
struct File {
    char f_name[MAXNAMELEN];  //文件的名字
    uint32_t f_size;  //文件的大小
    uint32_t f_type;  //文件的类型，有普通文件(FTYPE_REG)和目录(FTYPE_DIR)两种
    uint32_t f_direct[NDIRECT];  //文件的直接指针，每个文件控制块设有10个直接指针，用来记录文件的数据块在磁盘上的位置。
    uint32_t f_indirect;  //文件的间接指针，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。

    struct File *f_dir; // 指向文件所属的文件目录
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];  //占位符，让整数个文件结构体占用一个磁盘块
} __attribute__((aligned(4), packed));
```

Thinking 5.7

同步消息，用黑三角箭头搭配黑实线表示。同步的意义：消息的发送者把进程控制传递给消息的接收者，然后**暂停活动**，**等待**消息接收者的回应消息。

返回消息，用开三角箭头搭配黑色虚线表示。返回消息和同步消息结合使用，因为**异步消息**不进行等待，所以不需要知道返回值。

我们的进程通过和 file_server 这个进程的通信，来操作文件。

在用户进程中，通过 user/file.c 中的函数操作文件系统，在这些用户接口函数中，调用了 user/fsipc.c 中的函数，从而通过 user/fsipc.c 中的这些函数与文件系统进行了通信。在文件系统进程中，初始化完成后将运行 serve 函数，在这个函数中，调用了 ipc_recv， 通过返回值的不同(这些返回值定义在 include/fs.h 中)，在 switch...case 语句块中跳转到不同的函数，从而完成通信。

难点分析

1.设备驱动的内存映射

在我们的实验中，要实现的设备驱动主要有2种：即 console和 IDE disk。在内存中的布局如下表所示。

device	start addr	length
console	0x180003F8	0x20
IDE disk	0x180001F0	0x8

- 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 即为磁盘，用于存储文件等。

在用户态对上述外部设备进行读写等操作时，需要用到系统调用，在本次实验中实现的系统调用为 `syscall_write_dev` 与 `syscall_read_dev`。这两个函数实现的是将某段内存中的信息拷贝到外部设备的相应内存区域或将外部设备内存中的信息拷贝到某段内存中。

需要注意的是，在将物理地址与内核虚拟地址之间的转换时，需要将物理地址加上 kseg1 的偏移值而不是 kseg0；

并且，在判断物理地址合法性时，右边界应该写为 $\leq$！！！

`(pa >= 0x180003F8 && pa + len <= 0x18000418) && (pa >= 0x180001F0 && pa + len <= 0x180001F8)`

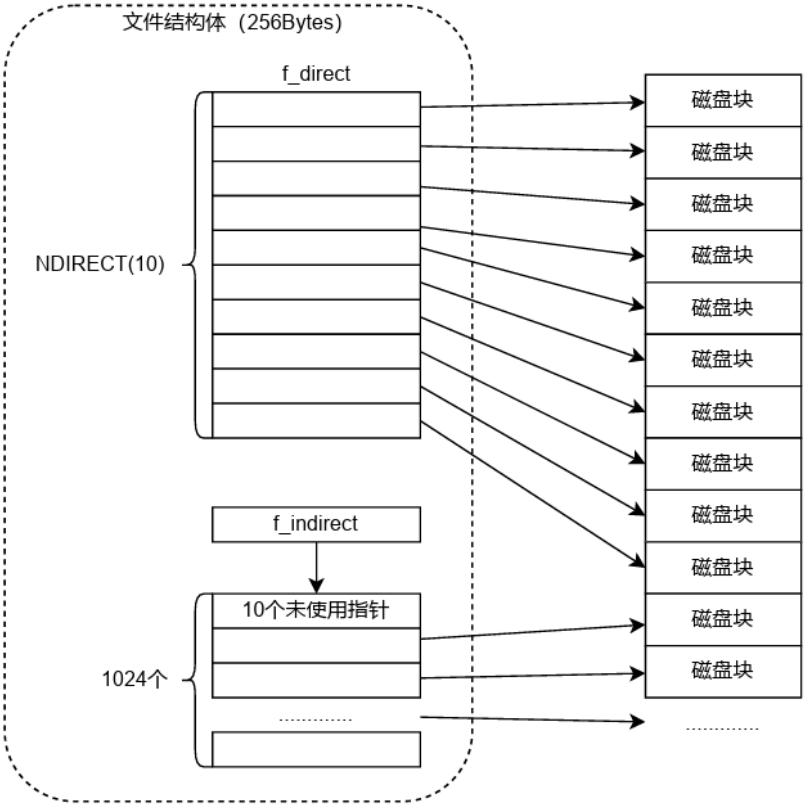
2.文件系统数据结构

对于MOS操作系统，我们使用文件控制块（File结构体）来管理文件：

```
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[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)];
} __attribute__((aligned(4), packed));
```

并以磁盘块作为基本储存单位：



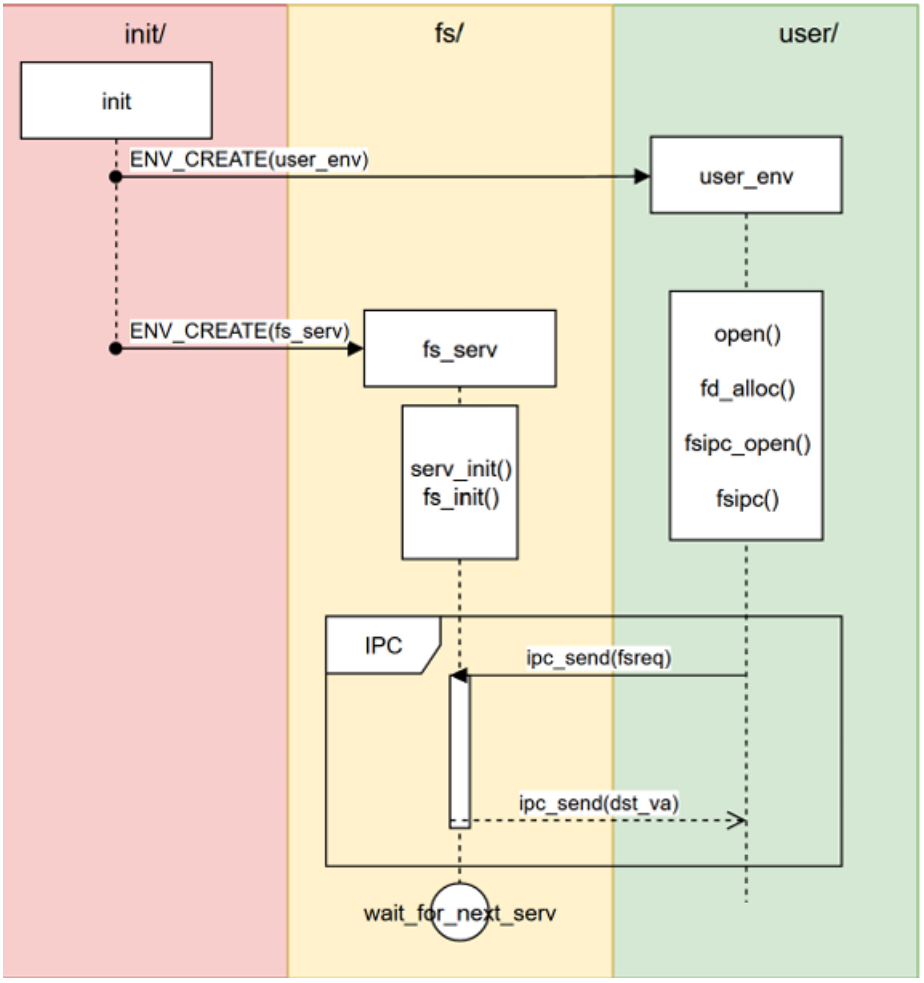
3.文件系统的用户接口

> 用户程序在发出文件系统操作请求时，将请求的内容放在对应的结构体中进行消息的传递，fs_serv 进程收到其他进行的 IPC 请求后，IPC 传递的消息包含了请求的类型和其他必要的参 数，根据请求的类型执行相应的文件操作（文件的增、删、改、查等），将结果重新通过IPC反 馈给用户程序。

请求类型包括：

```
enum {
    FSREQ_OPEN,
    FSREQ_MAP,
    FSREQ_SET_SIZE,
    FSREQ_CLOSE,
    FSREQ_DIRTY,
    FSREQ_REMOVE,
    FSREQ_SYNC,
    MAX_FSREQNO,
};
```

文件系统服务时序图如下：



实验体会

本次lab代码填写部分相对于lab4有了减少，但是代码阅读量较大大，需要认真阅读，理解调用逻辑。

lab5 的内容综合性较强，将前面几个 lab 的知识体系串联在一起，从 lab2 的内存结构到 lab4 的 IPC，每一步都需要深入理解才能帮助我们顺利完成 lab5 的实验。更多的精力要放在用户调用接口的逻辑上。

总而言之，lab5是倒数第2次实验了，离成功就差最后一步！