BUAA-OS-Lab5

一、思考题

Thinking 5.1

> 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。这是 一种错误的行为，在实际编写代码的时候这么做会引发不可预知的问题。请思考：这么做 这会引发什么问题？对于不同种类的设备（如我们提到的串口设备和IDE磁盘）的操作会 有差异吗？可以从缓存的性质和缓存更新的策略来考虑。

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。可能会导致数据不一致，如果对设备的写操作被缓存，那么后续读操作可能会从缓存中读取旧数据，而不是设备上的最新数据。

有差异，串口设备要求较高的实时性，如果写操作被缓存，可能导致数据发送延迟，影响系统正常运行；IDE磁盘数据即时性的要求不高，但错误的缓存行为可能导致文件系统损坏或数据丢失。

Thinking 5.2

> 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？一个目录下最多能有多少个文件？我们的文件系统支持的单个文件最大为多大？

一个磁盘块中最多能存储16个文件控制块，一个目录最多有16384个文件，单个文件最大为4MB。

Thinking 5.3

> 请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最 大磁盘大小是多少？

1GB

Thinking 5.4

> 在本实验中，fs/serv.h、user/include/fs.h 等文件中出现了许多宏定义， 试列举你认为较为重要的宏定义，同时进行解释，并描述其主要应用之处。

> #define SECT_SIZE 512 扇区大小512字节
>
> #define SECT2BLK (BLOCK_SIZE / SECT_SIZE) 1 个磁盘块是 8 个扇区#define DISKMAP 0x10000000
>
> #define DISKMAX 0x40000000 缓冲区地址0x10000000-0x40000000
>
> #define BLOCK_SIZE PAGE_SIZE 一个磁盘块大小为4KB
>
> #define FILE_STRUCT_SIZE 256 文件控制块大小256字节
>
> #define FILE2BLK (BLOCK_SIZE / sizeof(struct File)) 1个磁盘块有16个文件控制块
>
> #define MAXNAMELEN 128 最大文件名长度为128

Thinking 5.5

> 在Lab4“系统调用与fork”的实验中我们实现了极为重要的fork函数。那 么fork前后的父子进程是否会共享文件描述符和定位指针呢？请在完成上述练习的基础上 编写一个程序进行验证。

会共享文件描述符和定位指针。

```
int r, fdnum, n;
char buf[200];
fdnum = open("/newmotd", O_RDWR);
if ((r = fork()) == 0) {
    n = read(fdnum, buf, 5);
    printk("[child] buffer is \'%s\'\n", buf);
} else {
    n = read(fdnum, buf, 5);
    printk("[father] buffer is \'%s\'\n", buf);
}
```

Thinking 5.6

> 请解释File, Fd, Filefd结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。

```
struct Fd { // 文件描述符，该结构体用于管理文件描述符，存储相应数据
    u_int fd_dev_id; //  设备id
    u_int fd_offset; // 偏移量，类似于文件定位指针，标记随机读写的位置
    u_int fd_omode; // 用户对该文件的操作权限和方式
};
```

```
struct Filefd { // 已经打开的文件的文件描述符， 相较于Fd，有文件控制块f_file和fileid
    struct Fd f_fd; // 文件描述符，记录打开文件的部分信息
    u_int f_fileid; // 打开的文件的编号
    struct File f_file; // 文件控制块
};
```

```
struct File { // 文件控制块，包含文件的基本信息
    char f_name[MAXNAMELEN]; // filename文件名
    uint32_t f_size;     // file size in bytes文件大小
    uint32_t f_type;     // file type文件类型
    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; // 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; // the pointer to the dir where this file is in, valid
only in memory. 指向这个文件所在目录的指针
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)]; // 为了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。
} __attribute__((aligned(4), packed));
```

Thinking 5.7

> 图 5.9 中有多种不同形式的箭头，请解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。

图 5.9 以 UML 时序图的形式在宏观层面上展示了一个用户进程请求文件系统服务的过程(以 `open` 为例)。其中 `user_env` 所加载的程序不仅可以是实验源码已给出的 `fstest.c`，也可以是其他以 `main` 为入口函数的用户程序，你可以通过这种方式对文件系统服务进行测试。其中 IPC 系统调用的细节请参考 Lab4 的相关内容。(三种颜色不仅区分三个不同的程序，也表示进程执行的代码在我们的操作系统被载入内存前所处的文件位置)

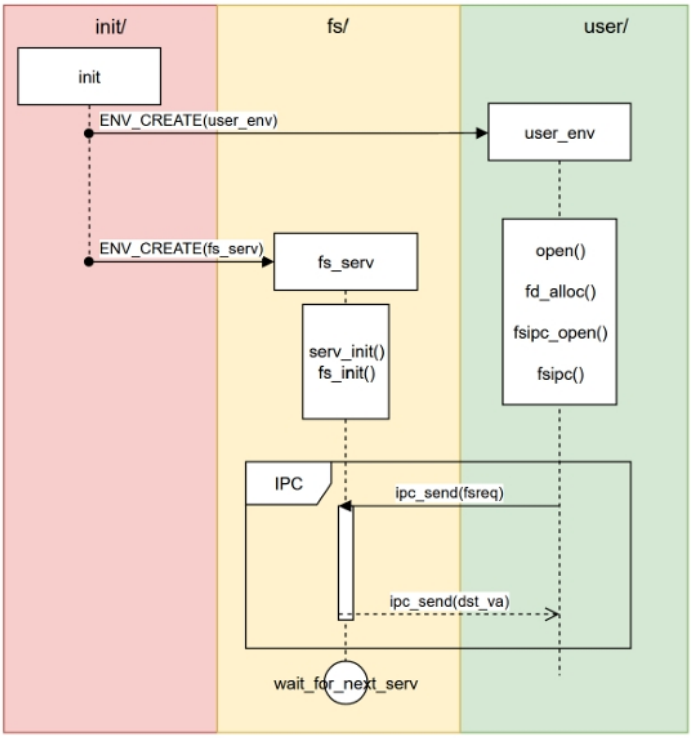

图 5.9: 文件系统服务时序图

ENV_CREATE(user_env)箭头：init/中创建用户进程，进程在user/中运行。

ENV_CREATE(fs_serv)箭头：init/中创建文件系统进程，进程在fs/中运行。

ipc_send(fsreq)箭头：用户进程执行打开文件操作，通过ipc将请AAA求发送给文件系统进程。

ipc_send(dst_va)箭头：文件系统进程收到请求后处理之后，向用户进程发送结果。

我们的操作系统是通过系统调用，将源地址的数据传输到目标地址中从而实现进程间的通信。

二、实验难点

难点1：理解目录、文件、磁盘块以及索引节点的关系

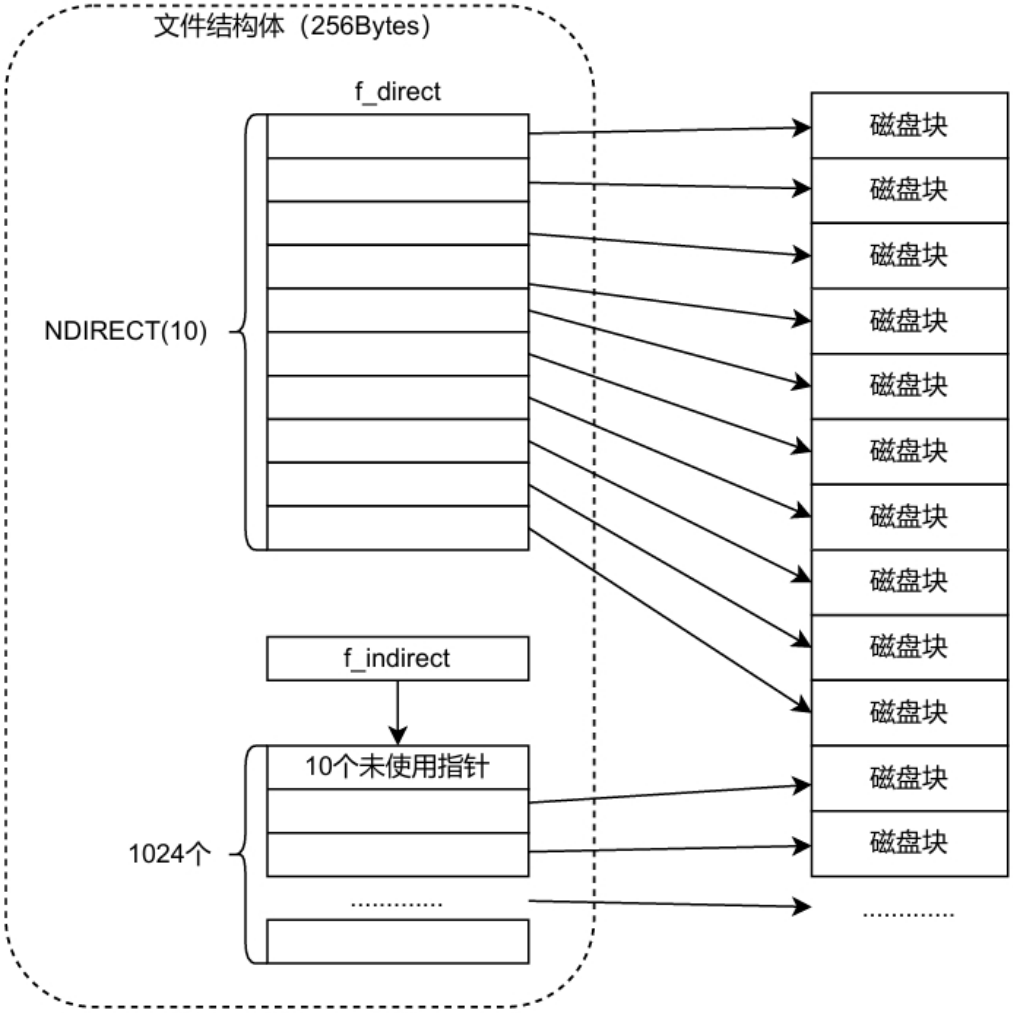

图 5.6: 文件控制块

- 目录是文件，文件目录是目录
- 文件中的数据保存在磁盘块中，可以通过直接索引获得文件信息，如果文件较大，则通过间接索引指向目录，再通过目录获得更多磁盘块用于存储文件信息。

难点2：理解文件系统服务进程如何服务用户进程

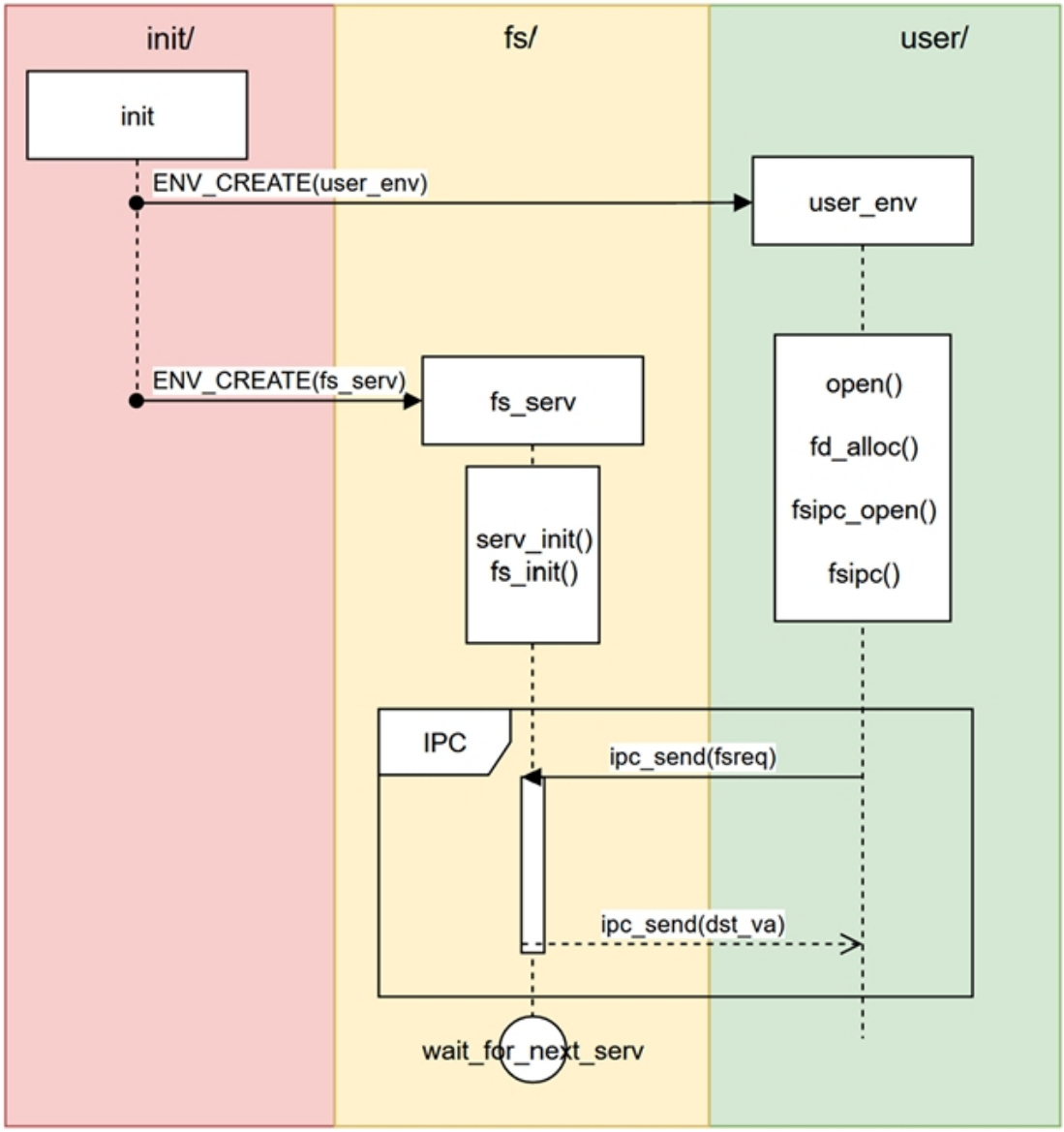

图 5.9: 文件系统服务时序图

用户进程需要通过进程间通信请求文件系统服务，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。

三、实验体会

在lab5我们实现了文件系统的构建，了解文件系统的基本概念和作用，掌握并实现文件系统服务的基本操作，其中运用的类似于面向对象中的继承和接口设计，底层向上层暴露封装好的接口，掩盖掉实现的细节的思想在我们后续的学习过程中有很大的意义，值得我们认真学习。