Lab5实验报告

思考题

T1

***查阅资料，了解 Linux/Unix 的 /proc 文件系统是什么？有什么作用？ Windows 操作系统又是如何实现这些功能的？proc 文件系统这样的设计有什么好处和可以改进的地方？ ***

proc文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。

作用：它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过proc得到系统的信息，并可以改变内核的某些参数。

在windows系统中，通过Win32 API函数调用来完成与内核的交互。

proc文件系统的设计的好处：将访问系统内核数据抽象为对文件的访问修改，简化了交互过程。

可以改进的地方：由于其长期驻留内存，会占据内存空间。

T2

***如果我们通过 kseg0 读写设备，我们对于设备的写入会缓存到 Cache 中。通过 kseg0 访问设备是一种错误的行为，在实际编写代码的时候这么做会引发不可预知的问题。请你思考：这么做这会引起什么问题？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刷新的策略来考虑。 ***

如果数据的写入先缓存到Cache中，待调出时才写回内存。那么该写入对于外设就会被滞后，导致串口设备如console这类实时交互型外设无法实时显示，正常工作。

T3

***比较 MOS 操作系统的文件控制块和 Unix/Linux 操作系统的 inode 及相关概念，试述二者的不同之处。 ***

MOS文件控制块和inode都包含了文件的基本信息，物理地址，大小等信息，都唯一对应一个文件。但是MOS的文件控制块在目录项中，而inode不在目录项中存放，而是统一存放在索引节点表中，目录项只存放inode的地址。

T4

***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？ ***

宏定义`BY2FILE=256`知，一个文件控制块的大小为256B，一个磁盘块大小为4KB，因此一个磁盘块最多能存储16个文件控制块。

一个目录最多可以指向1024个磁盘块，因此一共目录中最多1024*16=16384个子文件。

一个 文件的数据块最多有1024个，每个数据块可以存放4KB数据，因此一个文件最大可以为4MB

T5

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？

`DISKMAX`为0x40000000，即1GB，因此最大磁盘大小为1GB

T6

如果将 DISKMAX 改成 0xC0000000，超过用户空间，我们的文件系统还能正常工作吗？为什么？

不行，因为这样将磁盘块写入内存的过程中可能会覆盖掉操作系统内核原本的内容，导致操作系统崩溃。

况且由于文件系统服务是一个用户进程，访问高2G的虚拟地址空间会产生异常。

T7

在 lab5 中，fs/fs.h、include/fs.h 等文件中出现了许多结构体和宏定 义，写出你认为比较重要或难以理解的部分，并进行解释。

`fs/fs.h`：

```
//fs/fs.h
//MOS操作系统的disk磁盘只有一个，这里指数量为1，唯一的DISK的编号为0
#define DISKNO      1
//一个扇区的大小， bytes to sector
#define BY2SECT     512
//一个磁盘块包含8个扇区， sectors to block
#define SECT2BLK    (BY2BLK/BY2SECT)

//id为n的磁盘块(从0开始)，在内存中，被映射到虚拟地址空间的地址为DISKMAP+(n*BY2BLK)
#define DISKMAP     0x10000000

//我们可以处理的最大磁盘大小为1GB
#define DISKMAX     0x40000000
```

`include/fs.h`：

- 文件控制块

```
struct File {
    u_char f_name[MAXNAMELEN];  // filename，最大长度为128
    u_int f_size;           // file size in bytes
    u_int f_type;           // file type，分有普通文件 (FTYPE_REG) 和文件夹 (FTYPE_DIR)
    u_int f_direct[NDIRECT];//NDIRECT=10个文件的直接指针，是存放文件的数据块的编号，最大可以表示40KB的文件
    u_int f_indirect;//是一个间接磁盘块的编号，该磁盘块存放指向文件内容的磁盘块指针(int*类型)，共有1024个。不使用前10个
    struct File *f_dir;     // 指向文件所在目录，仅在内存中有效
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];//为了使FCB大小在256B设置的对齐数组， BY2FILE = 256
};
```

- 超块

```
struct Super {
    u_int s_magic;      // 魔数
    u_int s_nblocks;    // 磁盘上一共有多少磁盘块
    struct File s_root; // 根目录节点
};
```

- 文件系统请求

```
//表示不同的文件系统请求编号
#define FSREQ_OPEN  1
#define FSREQ_MAP   2
#define FSREQ_SET_SIZE  3
#define FSREQ_CLOSE 4
#define FSREQ_DIRTY 5
#define FSREQ_REMOVE    6
#define FSREQ_SYNC  7
//当用户进程需要请求文件系统服务时，请求信息分别根据请求类型保存在不同的结构体中。
struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
};

struct Fsreq_map {
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_set_size {
    int req_fileid;
    u_int req_size;
};

struct Fsreq_close {
    int req_fileid;
};

struct Fsreq_dirty {
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_remove {
    u_char req_path[MAXPATHLEN];
};
```

- 磁盘块：

```
//fs/fsformat.c
struct Block {
    uint8_t data[BY2BLK];//BY2BLK is 4096，一个单元为一个字节
    uint32_t type;       //块类型
} disk[NBLOCK];//NBLOCK is 1024
```

T8

阅读 user/file.c，你会发现很多函数中都会将一个 struct Fd* 型的 指针转换为 struct Filefd* 型的指针，请解释为什么这样的转换可行。

```
//user/fd.h
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

可以看出，`struct Filefd`的第一个元素就是`struct Fd`。由于c语言支持指针操作，而指针其实就是一个指向内存的地址，同一个文件的`struct Fd*`和`struct Fildfd*`本身地址相同，可以转化。

T9

在lab4 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？请在完成练习5.8和5.9的基础上编写一个程序进行验证。

会共享，而且两者页面的标识不是`PTE_COW`而是`PTE_LIBRARY`，验证代码如下：

在`user`目录下新建文件`mytest.c`，如下

```
#include "lib.h"
void umain() {
    int fd = open("/motd", O_RDWR);
    char buf[128];
    user_bzero(buf, 128);
    read(fd, buf, 10);
    if(fork() == 0) {
        //son
        char bufSon[128];
        user_bzero(bufSon, 128);
        read(fd, bufSon, 10);
        writef("son's buf is %s\n", bufSon);
    }else {
        char bufFa[128];
        user_bzero(bufFa, 128);
        read(fd, bufFa, 10);
        writef("father's buf is %s\n", bufFa);
    }
}
```

该进程会打开磁盘根目录下的`motd`文件，该文件内容为

```
This is /motd, the message of the day.

Welcome to the MOS kernel, now with a file system!
```

再在`user/Makefile`的`all:`后增加`mytest.x`和`mytest.b`。

在`init/init.c`加`ENV_CREATE(user_mytest)`。运行，可以得到输出

```
father's buf is otd, the m
son's buf is essage of
```

可以看出，父子进程共享文件描述符和定位指针，原因在于，当用户进程执行`open`操作时，文件系统进程会通过调用`serve_open`函数执行指令`ipc_send(envid, 0, (u_int)o->o_ff, PTE_V | PTE_R | PTE_LIBRARY);`,将`struct Filefd`结构体传递给用户进程的文件描述符页面，权限包括`PTE_LIBRARY`，为共享页面，因此父子进程也会共享文件描述符，定位 指针就在文件描述符结构体中，自然也被共享了。

T10

请解释Fd, Filefd, Open 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。

- 文件描述符：

```
// file descriptor
struct Fd {
    u_int fd_dev_id;//外设id
    u_int fd_offset;//当前的读写位置
    u_int fd_omode;//文件的打开方式
};
//以下为fd_omode的可取值：
#define O_RDONLY    0x0000      /* open for reading only */
#define O_WRONLY    0x0001      /* open for writing only */
#define O_RDWR      0x0002      /* open for reading and writing */
#define O_ACCMODE   0x0003      /* mask for above modes */

#define O_CREAT     0x0100      /* create if nonexistent */
#define O_TRUNC     0x0200      /* truncate to zero length */
#define O_EXCL      0x0400      /* error if already exists */
#define O_MKDIR     0x0800      /* create directory, not regular file */
```

文件描述符的作用是记录文件的打开信息和状态，用户可以直接根据文件描述符获取文件存储地址、向文件系统服务进程申请操作

- 记录文件详细信息的结构体

```
// file descriptor + file
struct Filefd {
    struct Fd f_fd;//文件描述符
    u_int f_fileid;//fileid表示文件的编号，根据fileid可以在opentab[1024]中唯一找到对应的struct Open，例如 struct Open *o = &opentab[fileid % MAXOPEN];
    struct File f_file;//文件控制块
};
```

由于文件描述符的信息过少，有时还需要通过fileid和FCB访问文件的打开信息和文件控制块相关的内容，所以需要该 结构体。

- 记录打开文件行为的结构体

```
struct Open opentab[MAXOPEN];
struct Open {
    struct File *o_file;    //被打开文件的FCB
    u_int o_fileid;         // file id
    int o_mode;             // open mode
    struct Filefd *o_ff;    // va of filefd page
};
```

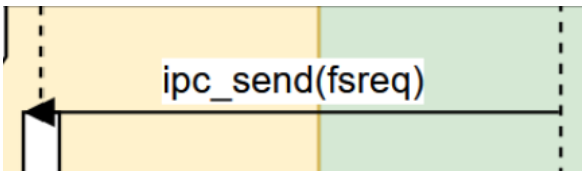
文件系统服务进程使用该结构体记录已打开的文件信息，通过fileid可以直接在opentab[]中直接检索到对应的struct Open。

T11

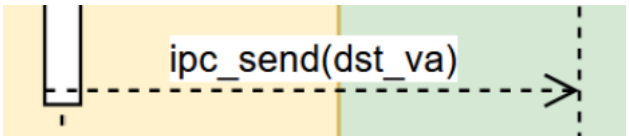
UML时序图中有多种不同形式的箭头，请结合UML 时序图的规范，解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。

共出现了三种箭头，都表示消息的传递。

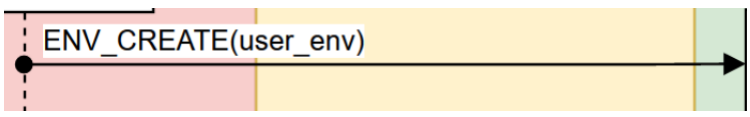
黑三角箭头搭配黑实线：同步消息，发送给文件系统进程请求



黑三角箭头搭配黑色虚线: 表示接受到请求后，返回消息



黑色三角箭头搭配黑实现加黑圆点：表示创建进程。



我们的操作系统实现进程的通信就是通过IPC进程间通信机制，发送进程调用`void ipc_send(u_int whom, u_int val, u_int srcva, u_int perm)`将val和srcva页面传递给whom进程，接受进程提前要调用函数`u_int ipc_recv(u_int *whom, u_int dstva, u_int *perm)`，将接受到的页面映射到dstva地址，并返回发送进程发送过来的val值。具体到文件系统和用户进程而言，用户进程主要通过调用不同的`fsipc_*`函数完成不同的文件操

作，`fsipc_*`函数间接调用函数`fsipc`完成与文件系统的通信。此函数将申请操作的类型type和申请操作的结构体`fsreq`发送给文件系统，之后用户进程从文件系统接受信息到地址`dstva`中完成请求。

```
static int fsipc(u_int type, void *fsreq, u_int dstva, u_int *perm)
{
    u_int whom;
    ipc_send(envs[1].env_id, type, (u_int)fsreq, PTE_V | PTE_R);
    return ipc_recv(&whom, dstva, perm);
}
```

而文件系统一直处于后台运行，调用`ipc_recv`函数等待用户进程发出请求。

T12

阅读serv.c/serve函数的代码，我们注意到函数中包含了一个死循环for (;;) {...}，为什么这段代码不会导致整个内核进入panic 状态?

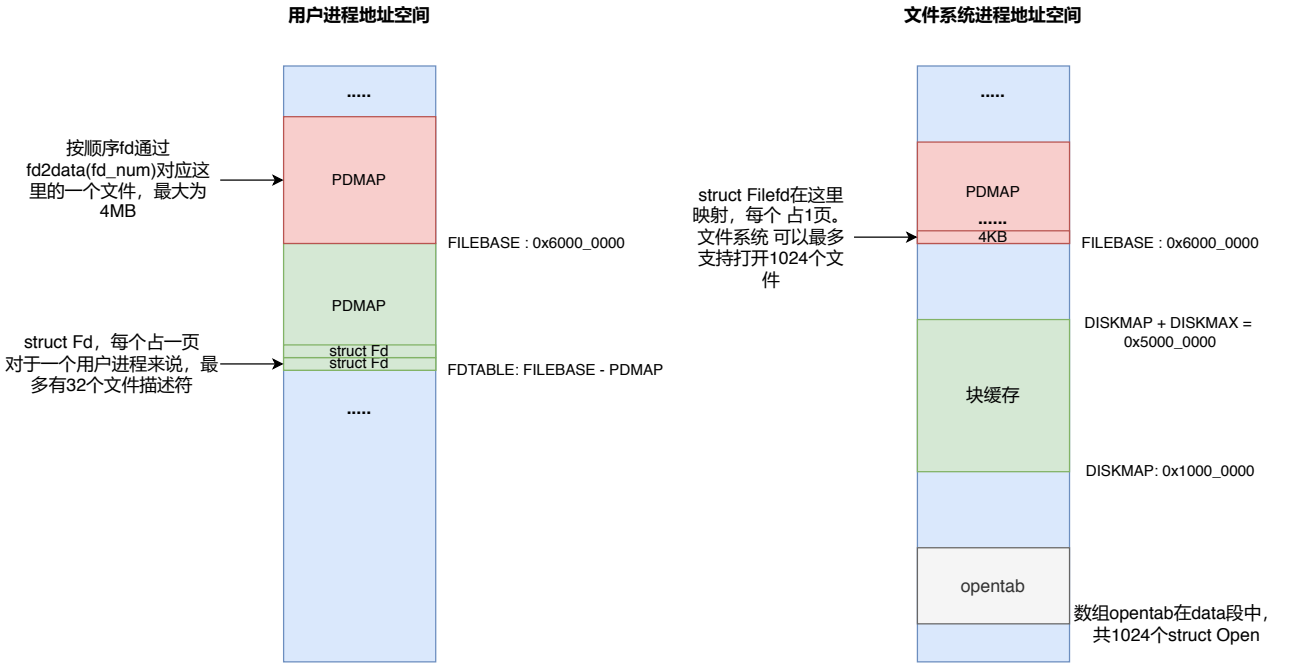
该函数是文件系统进程执行的核心函数，每次for循环时要执行`req = ipc_recv(&whom, REQVA, &perm);`，若用户进程未申请文件系统服务，则会被阻塞。只有用户进程申请文件系统服务时才会执行，执行完毕又会被阻塞。因此不会让内核进入panic状态。

实验难点

这次实验的主要难点主要集中在各个函数的调用关系，以及Lab5究竟做了哪几部分东西。

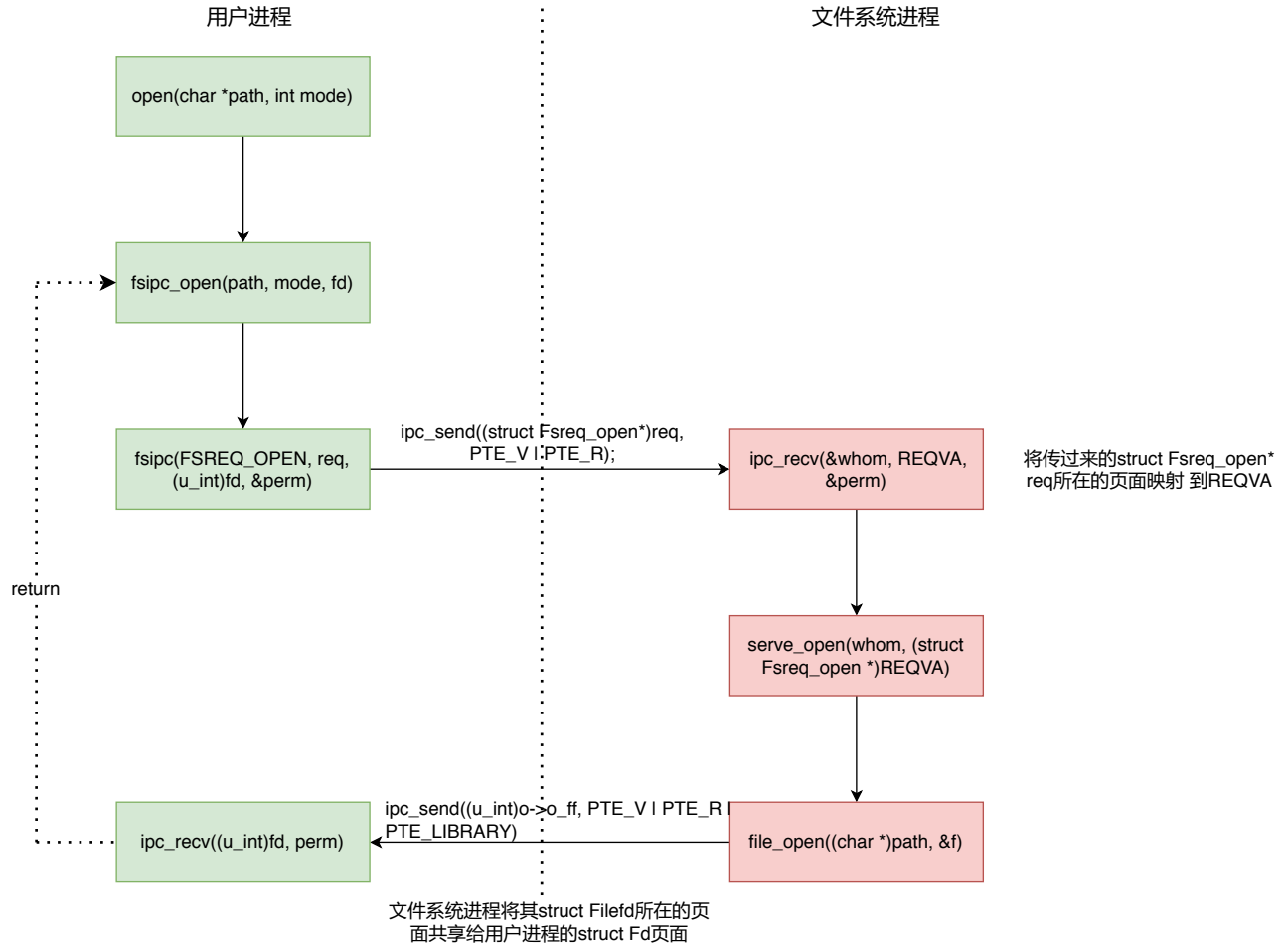
难点图示

用户进程和文件系统进程地址空间示意图



用户进程申请文件系统服务步骤

以 `open` 函数为例，当用户进程执行 `open` 函数后，



实验感想

Lab5学的是真的难，光一个文件代码量就1000行左右，函数多，调用关系也多。之前的lab重复学两遍左右就可以基本掌握，而这次学了3次以上才有了大概印象和头绪。好在exam和extra这两次上机并不难。