Lab5实验报告

一、实验思考题

Thinking 5.1

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？

Linux/Unix 系统上的 /proc 目录是一种文件系统，它以文件系统的方式为用户提供访问系统内核数据的操作接口。作为一种伪文件系统（也即虚拟文件系统），它只存在于内存当中，因此它会在系统启动时创建并挂载到 /proc 目录，在系统关闭时卸载并释放，是一种内核和内核模块用来向进程（process）发送信息的机制。

在windows系统中，通过Win32 API函数调用来完成与内核的交互。

好处：每次文件基本操作无需系统调用陷入内核，操作更方便快速

缺点：需要在内存中实现，占用内存空间；内核中的文件系统使内核更加庞杂

Thinking 5.2

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。

在kseg0段会和内核代码起冲突；对于设备文件，其相关I/O操作会触发中断，会使在cache上缓存的内容来不及写回产生错误。

Thinking 5.3

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：

文件控制块为File，将相关操作函数放入了Dev中

```
struct File {
     u_char f_name[MAXNAMELEN];  // filename
     u_int f_size;           // file size in bytes
     u_int f_type;           // file type
     u_int f_direct[NDIRECT];
     u_int f_indirect;
     struct File *f_dir;
     u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
 };

struct Dev {
     int dev_id;
     char *dev_name;
     int (*dev_read)(struct Fd *, void *, u_int, u_int);
     int (*dev_write)(struct Fd *, const void *, u_int, u_int);
     int (*dev_close)(struct Fd *);
```

```
    int (*dev_stat)(struct Fd *, struct Stat *);
    int (*dev_seek)(struct Fd *, u_int);
};
```

然后通过IPC完成文件操作。（用户进程和文件系统进程均在用户段kuseg，得靠系统调用通信，微内核嘛）

Unix/Linux操作系统：

仅将文件名留在了FCB中，将剩下索引无关的内容放入了索引结点inode中，这样一块磁盘能装更多的FCB，检索时方便载入内存

```
struct inode {
    ...
    unsigned long       i_ino;
    atomic_t            i_count;
    kdev_t              i_dev;
    umode_t             i_mode;
    nlink_t             i_nlink;
    uid_t               i_uid;
    gid_t               i_gid;
    kdev_t              i_rdev;
    loff_t              i_size;
    time_t              i_atime;
    time_t              i_mtime;
    time_t              i_ctime;
    ...
    struct inode_operations *i_op;
    struct file_operations  *i_fop;
    struct super_block      *i_sb;
    ...
};
```

Thinking 5.4

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？

```
#define BY2BLK      BY2PG               // 块大小和页大小相同，都是4096-4KB-0x1000
#define FILE2BLK    (BY2BLK / sizeof(struct File))
#define NBLOCK      1024                // 1024 blocks per disk
```

一个磁盘块最多存储$2^{12} \div 2^8 = 2^4 = 16$个文件控制块

一个目录下最多有$1024 \times 16 = 16384$个文件

支持单个最大文件：一个磁盘块能放$2^{12} \div 4 = 1024$个指针（4B一个），根据MOS机制，直接间接的指针加起来有1024个，每个指针指向一个磁盘块（4KB），所以最大$1024 \times 4KB = 4MB$

Thinking 5.5

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？

`0x4000_0000` 也即$2^{30}B = 1GB$

Thinking 5.6

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?

不能。由于MOS采用微内核思想将文件系统实现为用户进程，因而其不能访问内核空间

Thinking 5.7

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 */
#define DISKNO      1
#define BY2SECT     512                 // 512 byte per sector
#define SECT2BLK    (BY2BLK/BY2SECT)    // sectors per block
#define DISKMAP     0x10000000          // 文件系统进程地址空间中 块缓存位置
#define DISKMAX     0x40000000

/* include/fs.h */
struct File {
    u_char f_name[MAXNAMELEN];
    u_int  f_size;                      // 文件大小 byte
    u_int  f_type;
    u_int  f_direct[NDIRECT];           // 文件直接指针（块号）
    u_int  f_indirect;                  // 文件间接指针（块号）
    struct File *f_dir;                 // 指向这个文件所属的目录

    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 -4 -4];
};

struct Super {
    u_int s_magic;      // FS_MAGIC
    u_int s_nblocks;    // disk中总块数
    struct File s_root; // 根目录结点
}

/* fs/fsformat.c */
#define NBLOCK      1024                // 1024 blocks per disk
uint32_t nbitblock; // 表示位图占用了多少磁盘块
uint32_t nextbno;   // 下一个空闲块

struct Block {
    uint8_t data[BY2BLK];   // 块数据
    uint32_t type;          // 块类型
} disk[NBLOCK];
```

Thinking 5.8

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。

struct Filefd的首个成员即为struct Fd*，只需确保后续成员内容正确，强转Filefd*不过是告诉C这个指针域变大了而已，没有问题。

Thinking 5.9

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。

会共享文件描述符和定位指针；

程序可以使用user中的fdsharing.c

```
void
umain(int argc, char **argv)
{
    int fd, r, n, n2;

    if ((fd = open("motd", O_RDONLY)) < 0)
        user_panic("open motd: %e", fd);
    seek(fd, 0);
    if ((n = readn(fd, buf, sizeof buf)) <= 0)
        user_panic("readn: %e", n);

    if ((r = fork()) < 0)
        user_panic("fork: %e", r);
    if (r == 0) {
        seek(fd, 0);
        writef("going to read in child (might page fault if your sharing is
buggy)\n");
        if ((n2 = readn(fd, buf2, sizeof buf2)) != n2)
            user_panic("read in parent got %d, read in child got %d", n, n2);
        if(memcmp(buf, buf2, n) != 0)
            user_panic("read in parent got different bytes from read in child");
        writef("read in child succeeded\n");
        seek(fd, 0);
        close(fd);
        exit();
    }
    wait(r);
    //seek(fd, 0);
    if ((n2 = readn(fd, buf2, sizeof buf2)) != n)
        user_panic("read in parent got %d, then got %d", n, n2);
    writef("buf : %s\n",buf);
    writef("read in parent succeeded\n");
}
```

Thinking 5.10

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。

```
struct Fd { // 文件描述符
    u_int fd_dev_id;    // 设备的id
    u_int fd_offset;    // 当前的位置，用于fseek
    u_int fd_omode;     // 文件打开模式
};

struct Filefd {
```

```
    struct Fd f_fd;      // 文件描述符
    u_int f_fileid;      // 文件id
    struct File f_file; // 文件FCB (struct File)
};


struct Open {
    struct File *o_file;// 指向打开的文件FCB
    u_int o_fileid;      // 文件id
    int o_mode;          // 打开方式
    struct Filefd *o_ff;// 打开位置偏移量
};
```

Thinking 5.11

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。

粗箭头+实线：普通的同步消息。两个进程创建和ipc消息发送都是同步到，不发送成功就不阻塞着

细箭头+虚线：返回消息（reply），发出了ipc的请求，当然需要有返回的消息告知是否成功或返回想要的东西如fd文件描述符

Thinking 5.12

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?

因为在循环中，调用了ipc_recv()，在接收到相应请求信息之前，都会因为状态是ENV_NOT_RUNNABLE而被挂起不接受调度。

二、实验难点

1.定义一个文件系统操作服务

- user/fsipc.c：添加向文件系统发送请求的ipc操作 `fsipc_xx()`
- user/file.c：添加用户程序中使用接口 `xx()` （调用fsipc等具体请求通信函数）
- fs/fs.c：文件系统操作具体实现 `file_xx()`
- fs/serv.c：在文件系统进程中调用相应操作实现，提供服务 `serve_xx()`

2.整个文件系统结构

毕竟涉及好几个文件，好多结构体和函数，又杂又乱，看起来是费时费力的。

三、体会与感想

看代码真累（而且我现在也没看完，没用上就懒得看qwq），还没有完整图景，得下个决心。

1 “I/O 设备的物理地址是完全固定的，因此我们可以通过简单地读写某些固定的内核虚拟地址来实现驱动程序的功能”进一步验证了之前对内存管理的理解，借用内核段地址精确固定操控物理地址

2 还是有好些笔误：**实验正确结果**的2，应该是仅启动fs_serv进程；前面还有好些

3 fs_serv: 来自serv.c 还有类似user_testxx （中途的变化在哪里发生还没找到

4 init_disk() 按顺序初始化disk[0]、disk[1]、disk[2]... 逻辑会更通畅点吧，两个for也可和在一起

5 freeblock的测评似乎没有检测到nblock >= super->n_blocks的情况。

6 文件控制块direct、indirect说是指针，其实是块号bno，通过disk[bno]得到目标。希望说清，我误解疑惑了好一会儿。看到其他相关函数才意识到这点。

7 user/file.c的open函数，应该需要先syscall_mem_alloc再fsipc_map的吧