Lab5实验报告

19376273 陈厚伦

思考题

Thinking5.1

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 文件系统的设计有哪些好处和不足？

proc文件系统是伪文件系统，是服务于用户和内核的通信的。proc文件系统是一种无存储的文件系统，当读其中的文件时，其内容动态生成，当写文件时，文件所关联的写函数被调用。内核可以通过这个proc系统向用户空间提供接口来提供查询信息、修改软件行为。

Windows系统通过大量系统调用实现这样的功能，由用户根据需求调用这些系统调用，从而使用内核。

proc文件系统好处：proc系统将系统调用抽象为类似文件系统的形式，用户可以像操作文件系统一样，使用这些 “封装好的系统调用”。

proc文件系统不足：只有一个根目录/，不方便分类管理。

Thinking5.2

如果通过 kseg0 读写设备，那么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。

外设更新后对内存也进行了更新，但是cache中的没有被更新，从而导致错误。这种错误对磁盘出现概率较小，但是串口设备很容易发生。

Thinking5.3

一个磁盘块最多存储1024个指向其他磁盘块的指针，试计算，我们的文件系统支持的单个文件的最大大小为多大?

一个磁盘块大小是4KB，使用二级索引机制，第二级是一个磁盘块，总共可以有1024个索引，所以单个文件是1024 * 4KB = 4MB。

Thinking5.4

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?

一个文件控制块256B，一个磁盘块为4096B，一个磁盘块可以存储 4096/256=16 个文件控制块，一个目录文件最多指向1024个磁盘块，每个磁盘块可以装16个文件控制块，所以一个目录最多有1024*16=16384个文件。

Thinking5.5

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？

0x50000000-0x10000000 = 0x40000000 = 1GB.

Thinking5.6

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，我们的文件系统是一个用户进程，对内核的地址没有写的权限，所以不能正常工作。

Thinking5.7

阅读 user/file.c，思考文件描述符和打开的文件分别映射到了内存的哪一段空间。

文件描述符——FDTABLE——0x60000000-4MB = 0x5fc00000

打开的文件——FILEBASE——0x60000000

Thinking5.8

阅读 user/file.c，大家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。

在user/fd.h中定义了Filefd结构体：

```
// file descriptor + file
struct Filefd {
        struct Fd f_fd;
        u_int f_fileid;
        struct File f_file;
};
```

可以看到结构体Fd是结构体Filefd的第一个成员，两者具有相同的地址，所以指针可以进行直接转换。

Thinking5.9

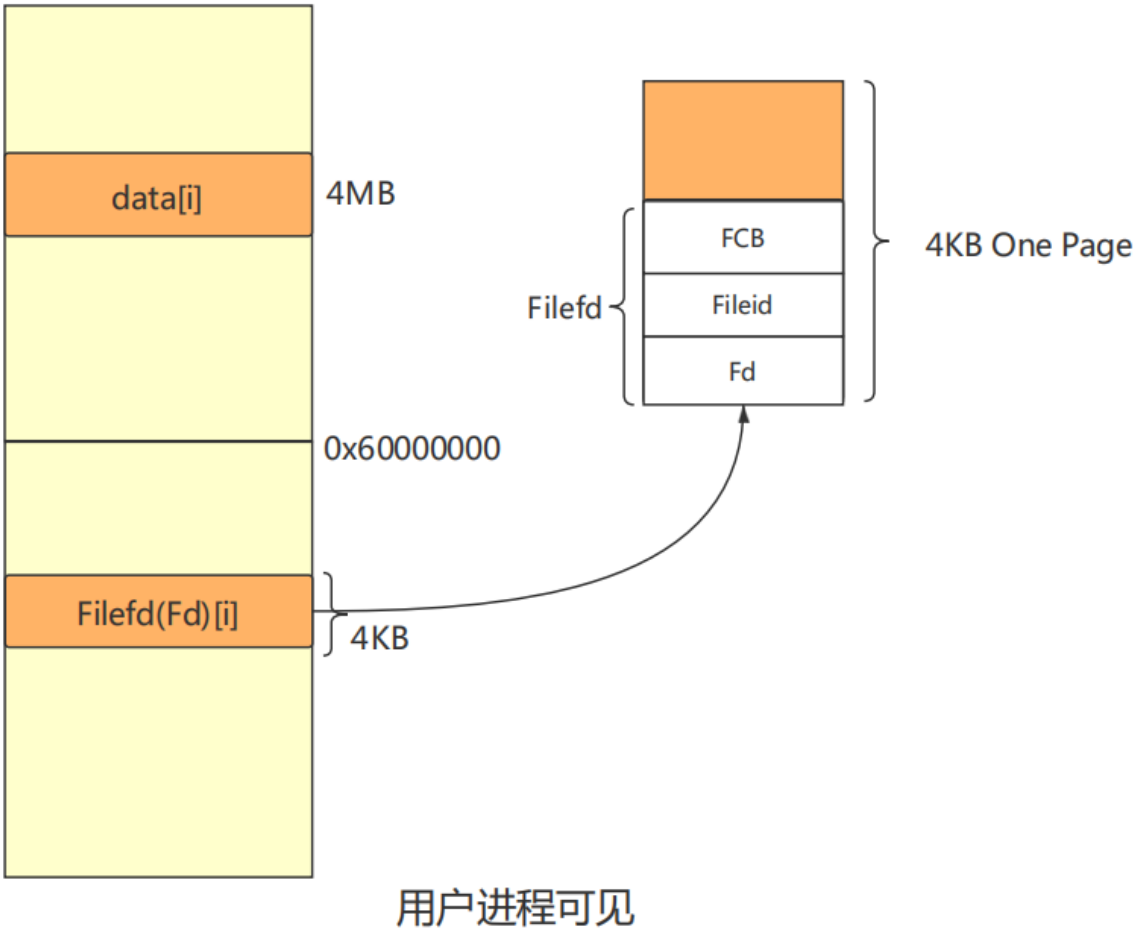
请解释 Fd, Filefd, Open 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。

```
// file descriptor
struct Fd {                 // 文件描述符 管理一个打开的文件 这个结构体是单纯的内存数据
        u_int fd_dev_id;    // 设备id
        u_int fd_offset;    // 类似于文件数据流指针的作用，seek，标记读写位置
        u_int fd_omode;     // 用户对该文件的操作权限
};

// file descriptor + file
struct Filefd {             // 文件描述符 描述打开的文件 比Fd多了FCB与fileid 是单纯的内存数据
        struct Fd f_fd;     // 文件描述符 记录打开的文件的部分信息
```

```
        u_int f_fileid;     // 文件系统为打开的文件的编号
        struct File f_file; // 文件的FCB
};

// struct Open
// 定义在fs/serve.c中 仅供文件系统进程使用
struct Open {               // 文件系统打开文件时 用来保存打开的文件信息 使用的时候会向
FIlefd中传递数据
    struct File *o_file;    // 指向FCB的指针
    u_int o_fileid;         // 文件系统为打开的文件的编号
    int o_mode;             // 用户对文件的操作权限
    struct Filefd *o_ff;    // 文件描述符Filefd的指针 地址
};
```



Filefd是以上图的方式来管理读入内存的文件的。

Thinking5.10

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?

```
void
serve(void)
{
        u_int req, whom, perm;

        for (;;) {
                perm = 0;
                req = ipc_recv(&whom, REQVA, &perm);
```

```
                // All requests must contain an argument page
                if (!(perm & PTE_V)) {
                        writef("Invalid request from %08x: no argument page\n",
whom);
                        continue;
                }

                switch (req) {
                        case xxx:
                                serve_XXX(whom, (struct Fsreq_xxx *)REQ_XXX);
                                break;
                        ...
                }

                syscall_mem_unmap(0, REQVA);
        }
}
```

serve在进程通信中发挥作用，观察代码可以发现，serve执行了

```
req = ipc_recv(&whom, REQVA, &perm);
```

进一步阅读`ipc_recv`的源代码：

```
u_int
ipc_recv(u_int *whom, u_int dstva, u_int *perm)
{
        //printf("ipc_recv:come 0\n");
        syscall_ipc_recv(dstva);
        if (whom) {
                *whom = env->env_ipc_from;
        }

        if (perm) {
                *perm = env->env_ipc_perm;
        }
        // dstva中可能存着被共享的物理页
        return env->env_ipc_value;
}
```

可以看到`ipc_recv`中调用了`syscall_ipc_recv(..)`，进一步阅读`sys_ipc_recv(..)`

```
void sys_ipc_recv(int sysno, u_int dstva)
{
        if (dstva >= UTOP) {
            return;
        }
        // 让一个进程进入等待接收的状态
        curenv->env_ipc_recving = 1;
        curenv->env_ipc_dstva = dstva;
        curenv->env_status = ENV_NOT_RUNNABLE;
        sys_yield();
}
```

当前serve执行到`ipc_recv`，通过`syscall_ipc_recv`进入使进程变成接收态(NOT_RUNNABLE)，并且被调度，直到它收到消息，变成RUNNABLE，才能够被调度，从`ipc_recv`的`syscall_ipc_recv`的后面的语句开始执行，并回到`serve`，所以这个服务中的文件系统进程不会死循环，空闲时会让出CPU让其余进程执行。

Thinking5.11

观察 user/fd.h 中结构体 Dev 及其调用方式。

```
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);
};
```

综合此次实验的全部代码，思考这样的定义和使用有什么好处。

对每一种设备的属性与处理函数进行结构体的封装，有“面向对象”的感觉。对于不同的设备通过访问设备结构体，就可以通过统一的方法获取设备的属性和调用设备的方法，实现了不同设备的“归一化”处理。

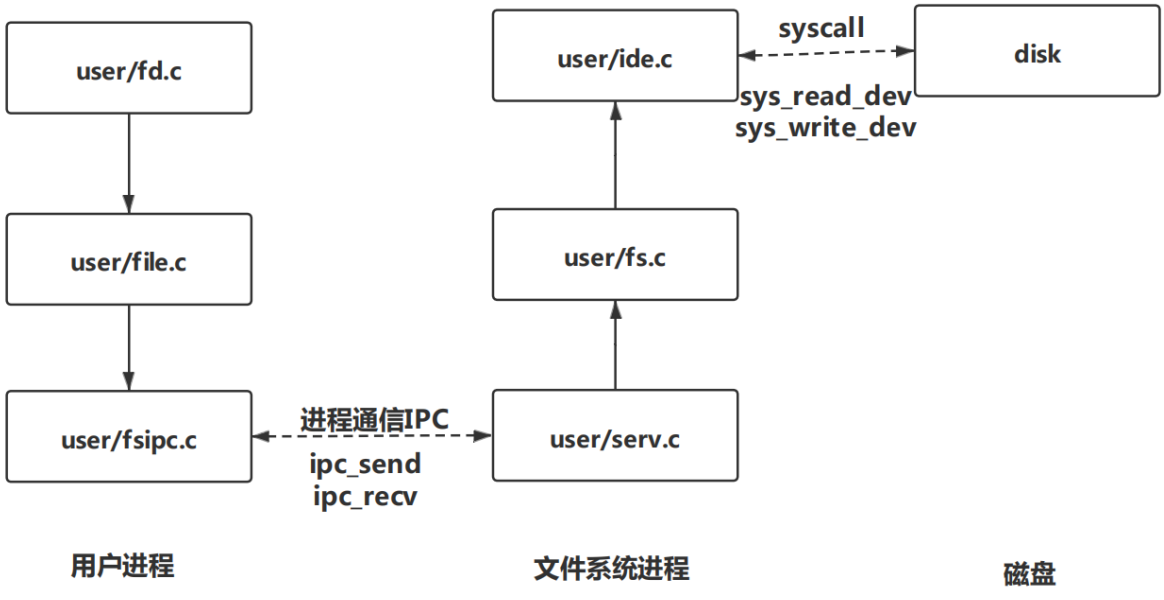
实验难点图示

> 文件系统层次关系梳理

文件系统可以分成四个层次：

文件系统用户接口、文件系统抽象层、文件系统具体实现、文件系统设备接口。

文件系统的逐级调用关系可以用下图表示：



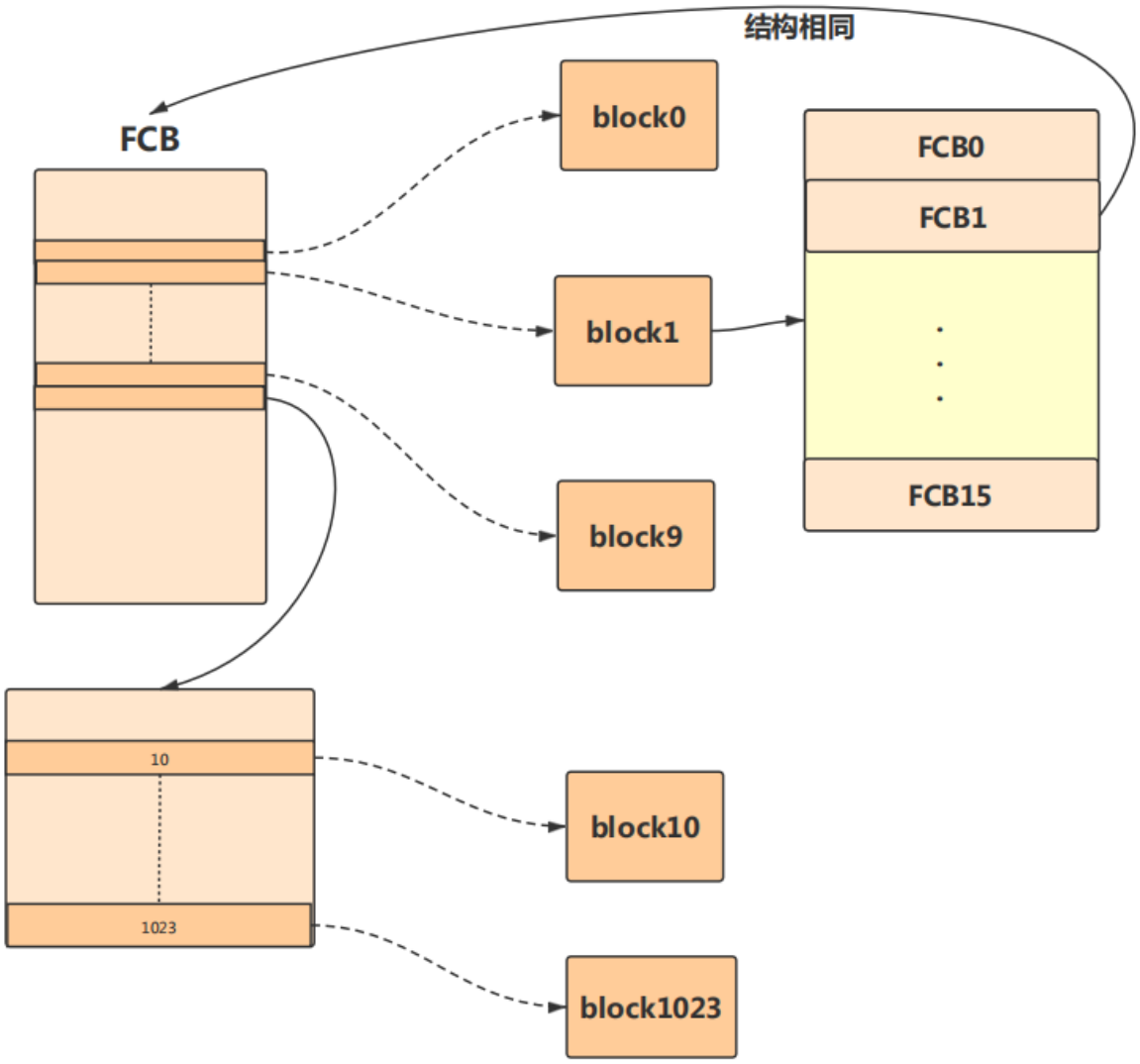
> 多级目录与多级索引

多级目录用来管理文件间的层次关系，多级索引用来管理单个文件的数据。两者并不是一个概念。

在我们的实验中，目录文件的内容是文件控制块`struct File`，目录文件的每一个数据块由16个文件目录项组成，找到相应的文件名，找到对应文件的FCB，进而找到该文件的数据块。

多级索引用来管理文件的数据块，我们的实验中使用二级索引机制。在FCB中可以有一级索引数组`f_direct[NDIRECT]`，和一个二级索引`f_indirect`，二级索引找到的磁盘块为索引块，这个磁盘块可以存储1024的FCB。

目录文件综合体现了这一点，以下图来说明：



> 文件系统的IPC机制

文件系统通过IPC进行用户进程与文件系统进程的通信，很好地体现了OS多进程的处理过程。

文件系统的通信过程如下所示：

用户在通过用户接口最终调用相应的`fsipc_xxxx()`函数，最终调用`fsipc()`将消息发送给文件系统进程。

文件系统进程中`serve()`函数调用`ipc_recv`接收用户传递来的信息，根据`req`类型调用相应的`serve_xxxx()`函数，对应的`serve_xxxx()`处理完毕后通过`ipc_send()`将信息发送给用户进程。

用户进程中最终又通过`ipc_recv()`接收文件系统进程发送回来的消息，最终完成一次完整的`ipc`通信过程。

> 设备驱动

设备驱动的难点其一在于要找到不同的设备寄存器对应的虚拟地址，其二是要对设备寄存器进行正确顺序的读写。

设备寄存器的虚拟地址分为三个部分：

```
/*  BASE:物理地址到虚拟地址的转换，本次实验中我们映射外设使用的是不经过cache的kseg1地址段，这个地址是0xa0000000
    PHY:设备的物理基地址，不同的设备有不同的物理地址，在ide磁盘中，这个地址是0x13000000
    OFFSET:设备不同的寄存器对应的物理地址偏移不同，具体情况需要查阅手册 */
```

虚拟地址通过`BASE + PHY + OFFSET`计算得到。

其次需要对设备寄存器进行合理顺序的读写，以写磁盘为例：

①将写入数据写入扇区数据缓冲区

②写入disk编号

③写入写的偏移

④写入“磁盘写”的标记，使磁盘开始写

⑤读出状态，判断写入操作是否成功

读磁盘与写磁盘过程类似。

体会与感想

lab5的需要写的代码量并不是特别大，但是需要自己阅读的代码量非常大。文件系统通过IPC机制进行用户进程与文件系统进程的交互是一个难点，花费了一整个晚上才将这个过程梳理明白。而且文件系统中有大量的结构体，各自发挥不同的作用，有不同的使用场景，比较难以区分，搞懂这些也花费了将近一个晚上的时间。文件系统的层次化非常好地将设备管理逐层抽象为一个个方便用户使用的接口，深刻体现了OS的抽象的强大作用，但是抽象的层次有些多，层次之间的调用关系比较复杂，细节很多，需要下真功夫才能够弄明白。

指导书反馈

应该是lab5的代码量非常大，所以在课程代码中找到了许多bug。由于自己的理解可能没有100%到位，所以可能有的bug是个人理解错所致，非真正的bug。

①`fd.c`中的`dup()`函数中的`if ((*vpd)[PDX(ova)])`最好改成`if ((*vpd)[PDX(ova)] & PTE_V)`,否则如果`PTE_V`无效时，但是这个目录项其余部分不为0时也可能会被判断为有效。

②`make_link_block()`函数中调用`save_block_link()`，由于`save_block_link()`中可能会再分配磁盘块作为索引块，可能导致磁盘块分配冲突。建议`make_block_link()`函数改成先执行`next_block()`记下返回值`v`，再利用得到的返回值`v`执行`save_block_link()`函数，最后返回`v`。

③代码中多次出现`f->file_size / BY2BLK`求文件的块数，由于文件的`size`只能以`BY2BLK`为单位增长减少，所以这样计算没有错误，但是为了严谨，最好改写成向上取整，即`(f->file_size + BY2BLK - 1) / BY2BLK`。

残留难点

lab5的代码量确实非常大，许多代码只是大概阅读了一下结构，并没有仔细研究其中的细节，对于文件系统的框架有了一定的熟悉，但是细节掌握的还不是很扎实，还需要之后多多阅读代码，加深理解与体会。