Lab 5 实验报告

15061200 李明轩

一，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？ ■

/proc 文件系统是一种内核和内核模块用来向进程 (process) 发送信息的机制，是一种虚拟的文件系统。这个伪文件系统让你可以和内核内部数据结构进行交互，获取有关进程的有用信息，在运行中 (on the fly) 改变设置 (通过改变内核参数)。而类似地，windows 中通过 Snapshot 机制（本质上也是创建并读取了一个“文件”的内容来获取运行信息），可以获取当前指定进程的（也可包括系统全部进程）的运行状况。在 proc 文件系统中，内核中进程的相关信息都作为一个个虚拟的文件存储在了内存中，通过统一调用信息传递和文件内容读写的方法，从而**方便有效地**完成获取进程信息和改变内核相关参数的操作。

> Thinking 5.2 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？ ■

根据代码中的宏函数，得知 DISKMAX 为 0xc000 0000 也即 3GB。

> Thinking 5.3 一个 Block 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？ ■

```
// Number of (direct) block pointers in a File descriptor
#define NDIRECT     10
#define NINDIRECT   (BY2BLK/4)
```

由于一个文件最多有 1024 个磁盘块，而一个盘块可以存储字节数为 BY2PG = 4KB

```
// Bytes per file system block - same as page size
#define BY2BLK      BY2PG
```

所以一个文件最大大小为 1024*BY2PG = 4MB

> 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？一个目录最多能有多少个子文件？ ■

在代码中找到如下定义

```
#define FILE2BLK    (BY2BLK/sizeof(struct File))
```

```
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;     // valid only in memory
    u_char f_pad[256 - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

因此，可以知道一个磁盘块中最多存储 FILE2BLK 个文件，一个目录最多能有 1024*FILE2BLK 个子文件

> Thinking 5.5 阅读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? ■

这段代码只是进程信息交互中的一部分，只要没有收到信号，就会阻塞，一直等待 ipc_recv，

```
for (;;) {
    perm = 0;

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

直到对应的进程发出了信息（也即需要文件系统运行，其他进程使用了某个文件系统的相关功能），文件系统进程才会继续运行，所以不会因为持续死循环而 panic。

> Thinking 5.6 阅读 user/file.c 中的众多代码，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。 ■

首先，这段代码转换的 struct Fd*类型指针是用户进程在打开文件后从文件系统返回的一个文件描述符指针，而这个指针所代表的物理页中的数据，最初来自于 opentab[i]结构体中的 o_ff，

进一步地，为了知道 o_ff 这个指针也即 struct Filefd*类型指针对应的地址代表的数据是什么，我们最早可以追溯到这段数据产生的时间节点，也就是文件打开表初始化的时候，代码如下，

```
// Initial array opentab.
for (i = 0; i < MAXOPEN; i++) {
    opentab[i].o_fileid = i;
    opentab[i].o_ff = (struct Filefd *)va;
    va += BY2PG;
}
```

在随后文件系统打开文件的过程中，o_ff 结构体被不断填充完整，如下

```
// Fill out the Filefd structure
ff = (struct Filefd *)o->o_ff;
ff->f_file = *f;
ff->f_fileid = o->o_fileid;
o->o_mode = rq->req_omode;
ff->f_fd.fd_omode = o->o_mode;
ff->f_fd.fd_dev_id = devfile.dev_id;

ipc_send(envid, 0, (u_int)o->o_ff, PTE_V | PTE_R | PTE_LIBRARY);
```

并最终将 o_ff 和用户进程所请求的 Struct Fd*类型指针映射到同一个物理页上。

一方面来说这个 Struct Fd*指针现在指向的区域已经包含了 struct Filefd*类型的数据，另一方面来说，后期用户进程在使用这个指针的时候需要用到 struct Filefd*中的数据，比如此处就需要 fileid 来将文件系统缓存中的文件内容映射到用户进程中。

```
// Step 3: Set the start address storing the file's content. Set size and fileid correctly.
// Hint: Use fd2data to get the start address.
va = fd2data(fd);
ffd = (struct Filefd *)fd;
size = ffd->f_file.f_size;
fileid = ffd->f_fileid;
```

综上所述，这个类型转换是可行而且必须做的。

> Thinking 5.7 在打开一个文件的过程中，用户进程和文件系统进程映射到各自内存中的内容有何不同（从位置、内容等方面作答）？阅读 user/fsipc.c 其中有很多函数都会向文件系统发送 `fileid` ，试解释其作用。 ■

从整体用户进程调用文件系统打开文件的流程来看，打开一个文件大致可以分为以下几个阶段：

用户进程调用 Open 函数（file.c）→ 分配空闲文件描述符，并将虚地址和文件路径发送给文件系统，调用 fsipc_open 函数→serv_open 等，分配空闲文件打开表，打开文件，返回文件控制块并填充文件打开表的相关域→文件系统返回打开文件相关信息的物理页→用户进程将内容从文件系统缓存中映射回当前进程地址空间

在这个过程中，文件系统映射文件打开表的地址是 FILEVA

```
// Set virtual address to map.
va = FILEVA;

// Initial array opentab.
for (i = 0; i < MAXOPEN; i++) {
    opentab[i].o_fileid = i;
    opentab[i].o_ff = (struct Filefd *)va;
    va += BY2PG;
}
```

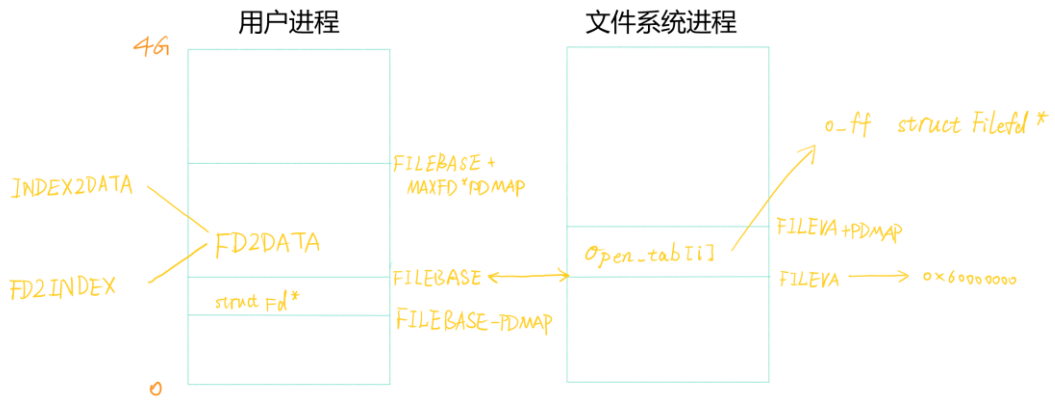
映射文件内容的起始地址是 DISKMAP。

而用户进程中，将文件系统返回的 Fd 类型指针存储在 FDTABLE 开始的一段大小为

PDMAP 的空间中，并最终将文件内容根据 fdindex 映射到 FILEBASE 起始，大小为 FDMAX*PDMAP 的一段空间中的特定位置。

```
#define FILEBASE 0x60000000
#define FDTABLE (FILEBASE-PDMAP)

#define INDEX2FD(i) (FDTABLE+(i)*BY2PG)
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)
```

整理出图示如下：



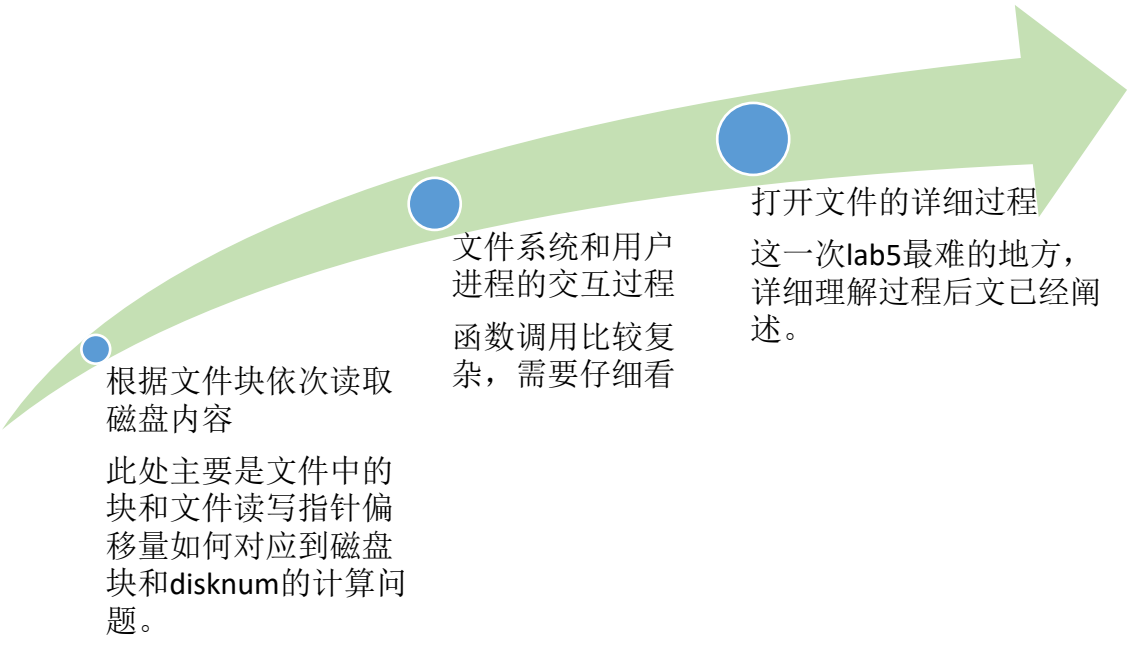
在整个过程中，fileid 起到了一个线索的作用，一方面文件系统打开文件后返回的关键参数就是经过哈希产生的 fileid 用来帮助用户进程映射到正确的文件，另一方面，用户进程为了保证打开文件和映射过程的正确性，需要使用 fileid 进行一系列校验，比如在 map 过程中，代码如下：

```
void
serve_map(u_int envid, struct Fsreq_map *rq)
{

    struct Open *pOpen;
    u_int filebno;

    void *blk;

    int r;

    if ((r = open_lookup(envid, rq->req_fileid, &pOpen)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
```

```
open_lookup(u_int envid, u_int fileid, struct Open **po)
{
    struct Open *o;

    o = &opentab[fileid % MAXOPEN];

    if (pageref(o->o_ff) == 1 || o->o_fileid != fileid) {
        return -E_INVAL;
    }

    *po = o;
    return 0;
}
```

根据用户进程回传的 fileid 反向哈希，找到原始的文件打开表内容，并对这一项中的 o_ff 引用次数进行检验（此时这个物理页应该同时有文件系统进程和用户进程在引用，引用次数至少为 2），同时检查文件打开表中的 fileid 和用户进程回传的 fileid 是否对应，如果以上检查通过，则说明没有问题，可以继续进行文件内容的映射。

二，实验难点图示



三，体会与感想

此次试验难度确实较大，虽然参考往届无数学长的经验可以快速完成，但是一知半解（即使是学长们也很少有理解透彻的）。尤其是文件打开的具体过程，对于前面几次 lab 的内容理解都是一次考验，包含了内存管理如不同进程地址空间虚拟地址对物理页的映射，进程调度的阻塞，进程交互的信息传递机制，这些都是理解这次文件系统的关键。在掌握了以上的基础之后，想要理解这次试验，还需要认真地阅读文件打开全部流程中的所有调用的函数，而且还需要仔细。

详细的文件打开过程在思考题中已经描述过，此处不再赘述，个人认为代码中最精彩的处理之一，就是在文件系统分配文件打开表项过程中对于 pageref 的应用：

```
// Find an available open-file table entry
for (i = 0; i < MAXOPEN; i++) {
    switch (pageref(opentab[i].o_ff)) {
    case 0:
        if ((r = syscall_mem_alloc(0, (u_int)opentab[i].o_ff,
            PTE_V | PTE_R | PTE_LIBRARY)) < 0) {
            return r;
        }
    case 1:
        opentab[i].o_fileid += MAXOPEN;
        *o = &opentab[i];
        user_bzero((void *)opentab[i].o_ff, BY2PG);
        return (*o)->o_fileid;
    }
}
```

关键就在于此处的 switch——case 语句没有 break，因此如果一页是完全空的，那么他就会先后经历被文件系统引用，然后继续 case1，哈希一个新的 fileid，填充相关域并返回 fileid。同理，如果一页已经使用了，我们也可以直接将它重新初始化并使用。

总的来说，这次实验应该是整个操作系统实验中最为精彩也最具有挑战性的一次，理解代码的过程也是一次个人思维的升华。

刷夜到凌晨一点，灵光一现的快感，此生难忘。

四，残留难点

根据代码中的注释，理论上应该有 3GB 的文件缓存，但是对照进程空间的相关参数，我认为文件最大达不到宏定义中的 3GB，否则会覆盖到其他的关键数据。举例来说，文件系统中的文件打开表的起始地址就是 FILEVA = 0x6000 0000， 明显是在 DISKMAP~DISKMAP+DISKMAX 范围内的，同样还有进程的相关参数如 env 结构体等等，这些都会冲突，所以我此处存疑。