杜佳男 21210110

操作系统lab5实验报告

1、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？

答：`/proc` 文件系统是一个虚拟文件系统, 它只存在内存当中，而不占用外存空间。`Linux` 内核提供了一种通过 `/proc` 文件系统,在运行时访问内核内部数据结构、改变内核设置的机制。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。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取 `proc` 文件时，`proc` 文件系统是动态从系统内核读出所需信息并提交的。`Windows` 通过提供相关的系统调用改变内核运行状态和查看进程信息。`proc` 文件系统这样的设计，好处是对系统调用进行了更多的抽象，并将其整合到了文件操作上，降低操作的复杂度；缺点是需要在内存中实现，占用内存空间。

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。

答：

- 当外部设备产生中断信号或者更新数据时，此时Cache中之前旧的数据可能刚完成缓存，那么完成缓存的这一部分无法完成更新，则会发生错误的行为。
- 不同种类的设备操作有差异。比如，串口设备读写更加频繁，信号多，在相同的时间内发生错误的概率远高于IDE磁盘。

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。

答：在`Unix/Linux`操作系统中，文件数据都储存在由多个扇区组成的"块"中，用来储存创建者、创建日期、大小等文件元信息的区域就叫做`inode`，每一个文件都有对应的`inode`，里面包含了与该文件有关的一些信息。可以用`stat`命令查看文件的`inode`信息，以`fs`文件夹为例：

```
git@21210110:~/21210110$ stat fs
  File: fs
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 72h/114d        Inode: 8128935     Links: 2
Access: (0775/drwxrwxr-x)  Uid: (  997/     git)   Gid: (  997/     git)
Access: 2022-06-04 06:11:28.870099280 +0800
Modify: 2022-06-04 06:10:49.593565794 +0800
Change: 2022-06-04 06:10:49.593565794 +0800
 Birth: -
git@21210110:~/21210110$
```

而`MOS`操作系统的文件控制块存储的内容和其有所不同，具体可看`Thinking 5.4`的图。在访问文件时，`Unix/Linux`操作系统通过`inode`号码识别文件，访问`inode`节点，读取文件信息，而在实验用的`MOS`操作系统中则是通过文件的直接指针和间接指针寻找文件所对应的磁盘块去查找文件。

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？

```
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;     // the pointer to the dir where this file is in, valid only in memory.
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

答：一个磁盘块大小为`4KB`，`f_pad`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，其大小为`BY2FILE`减去`File`结构体其他元素的大小，因此可以推断一个文件控制块大小为`BY2FILE`，也即`256B`。因此，1个磁盘块中最多能存储`4KB/256B=16`个文件控制块；单个文件有10个直接指针，最多1024个间接指针，因此一个目录下最多能有`1024*16=16384`个文件；单个文件最大为`1024*4KB=4096KB=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？

答：实验采用`kseg1`区域映射磁盘，因此最多处理`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？

答：不能。我们的文件系统是放在用户态而不是内核态，而用户态不能访问用户空间以上的内核空间，因此文件系统不能正常工作，此外，超过用户态的部分还可能覆盖掉内核内容，导致系统运行异常。

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。

答：

- 比较重要的结构体：
  - `struct File`，详见`Thinking 5.4`，`f_name`为文件名称，文件名的最大长度`MAXNAMELEN`值为 128。`f_size`为文件的大小，单位为字节。`f_type`为文件类型，有普通文件`(FTYPE_REG)`和文件夹`(FTYPE_DIR)`两种。`f_direct[NDIRECT]`为文件的直接指针，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。每个磁盘块的大小为`4KB`，这十个直接指针能够表示最大`40KB`的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。`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 Super`，`s_magic`为魔数，用于识别该文件系统，为一个常量；`s_nblocks`用来记录本文件系统有多少个磁盘块，本文件系统为 1024；`s_root`为根目录，其`f_type`为`FTYPE_DIR`，`f_name`为"/"。
- 比较重要的一些宏定义

```
#define BY2FILE     256
#define NDIRECT     10                  /*Number of direct point*/
#define NINDIRECT   (BY2BLK/4)          /*Number of indirect point*/
#define BY2BLK      BY2PG               /* Bytes per file system block*/
#define BIT2BLK     (BY2BLK*8)
#define BY2SECT     512                 /* Bytes per disk sector */
#define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */
```

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。

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

答：因为结构体`Filefd`中储存的第一个元素就是`struct Fd*`，因而对于相匹配的一对`struct Fd`和`struct Filefd`，两者的指针指向相同的虚拟地址，所以可以通过指针转化访问`struct Filefd`中的其他元素。

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。

答：在`fork`前打开的文件，`fork`后的父子进程可以共享打开的文件描述符。测试程序可参考lab5-2-exam的测试程序：

```
#include "lib.h"


void umain()
{
    int r, fdnum, n;
    char buf[200];
    fdnum = open("/newmotd", O_RDWR);
    if ((r = fork()) == 0) {
     n = read(fdnum, buf, 5);
     writef("[child] buffer is \'%s\'\n", buf);
    } else {
     n = read(fdnum, buf, 5);
     writef("[father] buffer is \'%s\'\n", buf);
    }
    while(1);
}
```

Thinking 5.10

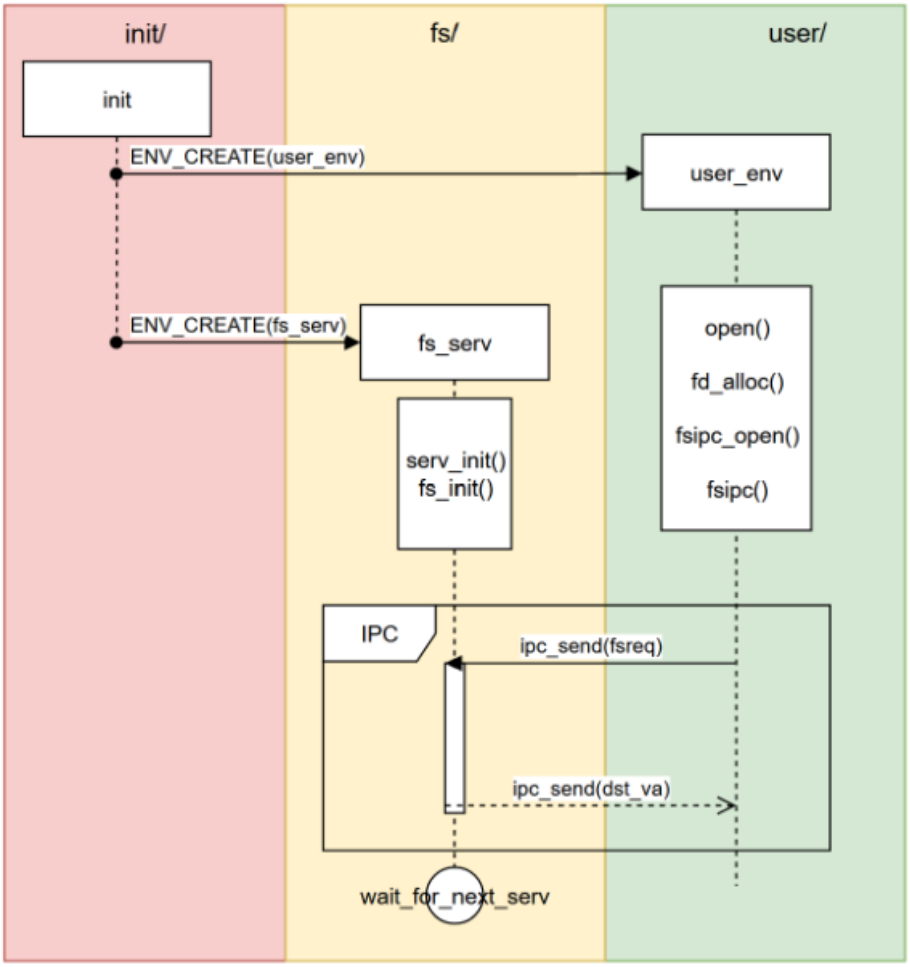
请解释Fd, Filefd, Open 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。

答：

- `Fd`结构体用于表示文件描述符，`fd_dev_id`表示文件所在设备的`id`，`fd_offset`表示读或者写文件的时候，距离文件开头的偏移量，`fd_omode`用于描述文件打开的读写模式。它主要用于在打开文件之后记录文件的状态，以便对文件进行管理/读写，不对应物理实体，只是单纯的内存数据。
- `Filefd`结构体是文件描述符 和 文件的组合形式，`f_fd`记录了文件描述符，`f_fileid`记录了文件的`id`，`f_file`则记录了文件控制块，包含文件的信息以及指向储存文件的磁盘块的指针，对应了磁盘的物理实体，也包含内存数据。
- `Open`结构体在文件系统进程用用于储存文件相关信息，`o_file`指向了对应的文件控制块，`o_fileid`表示文件`id`，用于在数组`opentab`中查找对应的`Open`，`o_mode`记录文件打开的状态，`o_ff`指向对应的`Filefd`结构体。

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。



答：在`UML`时序图中，实箭头代表提交信息，虚箭头代表返回信息，在如图所示的用户进程请求文件系统服务的过程中，首先在`init`函数中调用`ENV_CREATE(user_env)`和`ENV_CREATE(fs_serv)`，两者各自执行对应的函数，并在执行过程中进行`ipc`通信。用户程序`user_env`是通信发出方，用实箭头发往`fsreq`，在发出文件系统操作请求时，将请求的内容放在对应的结构体（`fsreq`）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（定义在`include/fs.h`中）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，用虚箭头表示。

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?

答：因为在实际调用`ipc_recv`函数，这个进程会等待其他进程向他发送请求文件系统调用再继续执行，如果没有接收到请求就会一直等待，因此不会自己执行死循环。

serve：

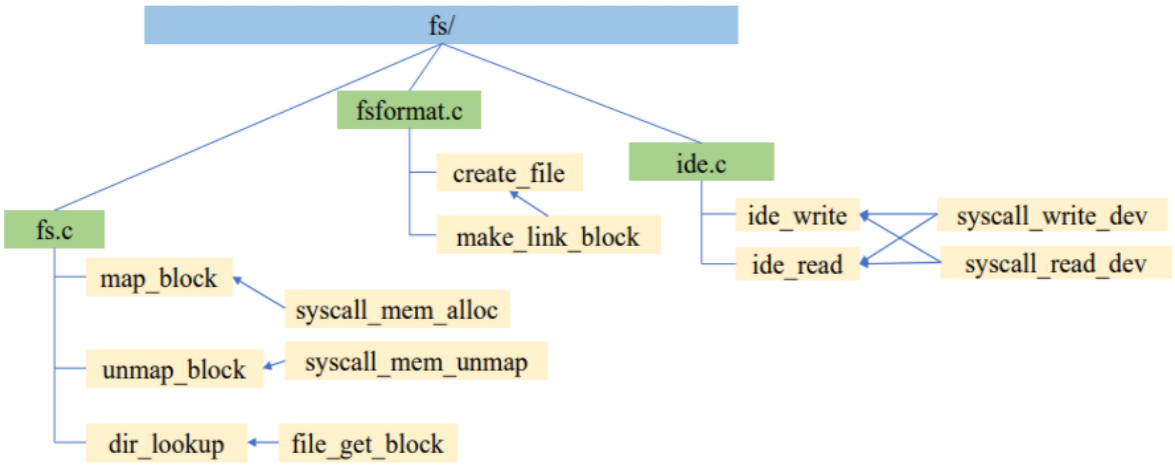
```
void
serve(void)
{
    u_int req, whom, perm;

    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);
```

sys_ipc_recv：

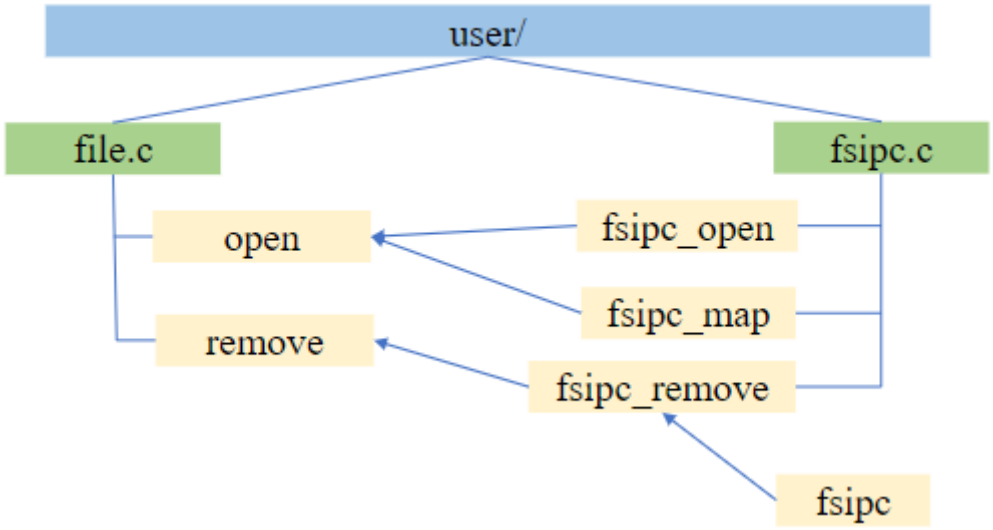
```
/*** exercise 4.7 ***/
void sys_ipc_recv(int sysno, u_int dstva)
{
    if (dstva >= UTOP) return -E_INVAL;
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sys_yield();
}
```

2、实验难点

- IDE磁盘的驱动：`ide_write`和`ide_read`
- 使用文件系统维护磁盘块的申请和释放
- 文件控制块的结构、其直接文件指针和间接文件指针的使用
- 块缓存，建立起磁盘地址空间和进程虚存地址空间之间的缓存映射



- 用户接口，向用户提供相关的接口使用



3、体会与感受

完成此实验首先需要对IDE磁盘的读写有一个大概的了解，参考内核态驱动完成对IDE磁盘读写函数的编写，其次需要明白文件控制块的相关内容，了解`File`结构体每一个成员的意义，熟悉直接文件指针和间接文件指针的使用，最后，还需要对用户接口进行编写，以完成用户态调用。实验整体内容难度较高，耗时约一周，课上Exam难度合理，Extra内容过多。

4、指导书反馈

- Exercise 5.5题面的512 bytes改为4096 bytes