# OS Lab-5 Report

## 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 文件系统的设计有哪些好处和不足？**

/proc 文件系统是一个虚拟文件系统，通过对这些虚拟文件的读写可以与内核中实体进行通信。具体能实现的功能包括读取系统数据、进程信息甚至修改系统参数等。Windows系统一般通过Windows API来实现类似的功能。这样的设计简化用户程序和内核空间的交互过程，更加方便快捷。

好处：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：需要在内存中实现，占用内存空间。

**Thinking 5.2 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。 这是一种错误的行为，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。**

对于写入操作：

在采用Write-back刷新策略时，写入数据只有在cache被换出时才会进行写回，导致后面的操作覆盖了前面操作，只进行最后一次操作。对串口设备，只有Cache刷新后才能看到输出，且只能看到最后一个字符。类似的，IDE磁盘可能只会写入最后一个扇区。

但如果采用Write-through策略进行刷新，CPU向Cache写入数据时，也会向内存相同地址也写一份。这样就避免了上面所说的问题，可以正常工作。

如果是读取操作：问题更大，任何一种策略都可能会读取到旧的、过时的数据，因此产生错误。

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。

对于串口设备来说，读写频繁，信号多，在相同的时间内发生错误的概论远高于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。**

文件系统中的inode：

文件储存在硬盘上，硬盘的最小存储单位叫做"扇区"（Sector）。每个扇区储存512字节（相当于0.5KB）。

操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的最小单位。"块"的大小，最常见的是4KB，即连续八个 sector组成一个 block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为"索引节点"。

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。

```
    * 文件的字节数
    * 文件拥有者的User ID
    * 文件的Group ID
    * 文件的读、写、执行权限
    * 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。
    * 链接数，即有多少文件名指向这个inode
    * 文件数据block的位置
```

**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？**

一个磁盘块最多存储16个文件控制块；单个文件最多有1024个指针，指向1024个磁盘块，所以一个目录下最多16384个文件。一个磁盘块大小为4KB。单个文件有10个直接指针，最多1024个间接指针，那么单个文件最大为((1024+4)*4KB=4112KB)

**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？**

根据我们小操作系统的代码，我们磁盘最大的大小不能超过DISKMAX，0x40000000字节，也就是1GB。

但是，如果但从系统架构所决定的最大可支持的磁盘大小角度考虑，0x10000000之下要存储一页ipc用的buffer，所以必须从DISKMAP，0x10000000开始映射缓存的硬盘块。缓存的硬盘块是在serv.c这个用户内存空间里的，而serv.c进程会从FILEVA，0x60000000开始，为Open结构分配空间。

一个正常的用户进程中，FDTABLE，也就是(FILEBASE-PDMAP)的位置是放置fd的。但是考虑到我们serv.c本身就是文件系统服务，不会也无法使用用户态提供的fd系列操作，且我们的Open结构的空间也覆盖了fd的Data区域，所以其实我们不需要考虑这个问题。

根据上面的结论，可以得出，serv.c可以被用来缓存磁盘块的大小的空间是0x10000000-0x60000000共0x50000000，1.25GB空间

**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，根据上面的分析，在大于0x50000000之后，就会覆盖掉Open结构进而可能出现潜在问题，达到0xC0000000, 超过用户空间之后更是会试图访问内核数据，会引发异常并panic。

**Thinking 5.7 在 lab5 中，fs/fs.h、include/fs.h 等文件中出现了许多宏定义，试列举你认为较为重要的宏定义，并进行解释，写出其主要应用之处。**

```
//include/fs.h
// Bytes per file system block - same as page size
#define BY2BLK      BY2PG
#define BIT2BLK     (BY2BLK*8)

// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN  128

// Maximum size of a complete pathname, including null
#define MAXPATHLEN  1024

// Number of (direct) block pointers in a File descriptor
```

```
#define NDIRECT     10
#define NINDIRECT   (BY2BLK/4)
#define MAXFILESIZE (NINDIRECT*BY2BLK)
#define BY2FILE     256

// File types
#define FTYPE_REG       0   // Regular file
#define FTYPE_DIR       1   // Directory

// File system super-block (both in-memory and on-disk)
#define FS_MAGIC    0x68286097  // Everyone's favorite OS class

// Definitions for requests from clients to file system
#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


//fs/fs.h
/* IDE disk number to look on for our file system */
#define DISKNO      1

#define BY2SECT     512 /* Bytes per disk sector */
#define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP     0x10000000

/* Maximum disk size we can handle (1GB) */
#define DISKMAX     0x40000000
```

**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。**

user/file.c里的struct Fd *指针都是open之后的，而open的过程中调用了fsipc_open函数，并将一个struct Fd型指针的值发送给serv。serv会用ipc将fd指针的所在页映射上一个struct Filefd。而Filefd的第一个元素就是一个Fd，因此转换之后不会出现问题。

**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？请在完成上述练习的基础上编写一个程序进行验证。**

fork后的父子进程可以共享打开的文件描述符。

```
#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);
}

/* expected output:
======================================================================
[father] buffer is 'This '
[child] buffer is 'This '
======================================================================
*/
```

**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。**

1. struct Fd定义在user/fd.h，是一个文件描述符结构，是库函数保存用户进程已打开文件使用的。
    - fd_dev_id：打开文件的id，也就是该文件描述符对应的抽象文件的实际类型
    - fd_offset：当前读/写的偏移值，也就是下一次操作从文件的哪个地方开始
    - fd_omode：当前文件打开的模式，只读/只写/读写等，可在判定操作是否合法时用。
2. struct Filefd定义在user/fd.h，是文件描述符+文件id+文件控制块的结构
    - f_fd：一个文件描述符。
    - f_fileid：对应于一个全局的文件编号，用来向文件系统请求服务。
    - f_file：对应文件的文件控制块。
3. struct Open定义在fs/serv.c，是文件系统服务用来保存整个系统的已打开文件的结构。
    - o_file：真实的，指向对应文件在硬盘块缓存上文件控制块的地址，用来对文件进行属性进行更改。
    - o_fileid：全局唯一的文件编号，和struct Filefd里的f_fileid对应。
    - o_mode：文件打开的模式，和struct Fd的fd_omode对应

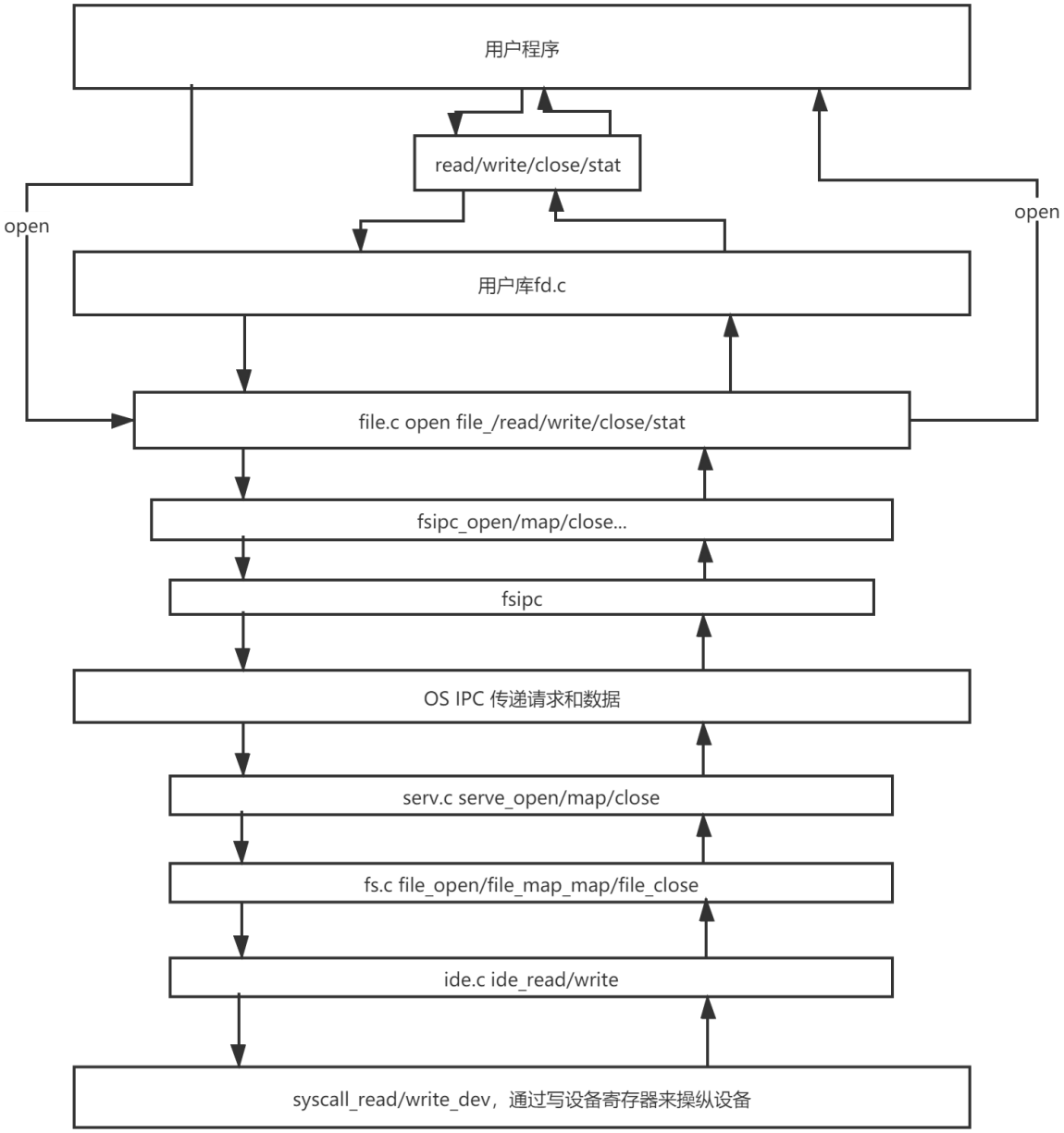
**Thinking 5.11 上图中有多种不同形式的箭头，请结合 UML 时序图的规范，解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。**

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`是通信发出者，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传递的信息包含了请求的类型和其他必要的参数，根据请求的类型执行相应的文件操作（文件的增删改查等），最后将IPC返回给用户程序。

**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?**

serve进程随着系统运行开始时开始执行，每次循环都调用了 ipc_recv，该进程会进入 NOT_RUNNABLE 状态，随时相应用户进程发出的文件请求，不会一直占用CPU，直到结束系统杀死进程。此进程为用户态进程，不会导致内核进程陷入panic。

# 二、实验难点图示




# 三、体会与感想

Lab-5的难度明显比之前有所增加，整个函数的调用关系非常复杂，要填写的代码内容虽然不多，但是要读的代码却很多。

Lab-5代码的理解就是把握住文件系统里的层层抽象，ide.c将将磁盘块的操作抽象出来；serv.c将磁盘抽象为一个目录树，利用ipc提供文件系统服务；file.c利用fsipc.c封装对于文件的种种操作；fd.c更进一层提供了对所有“虚拟文件类型”的访问操作。