**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**

**处和可以改进的地方？**

/proc文件系统是一个虚拟文件系统，是一种内核和内核模块用来向进程(process)发送信息的机制。通过这个伪文件系统，可以使用一种新的方法在Linux内核空间和用户空间之间进行通信。它只存在于内存空间，不占用外存空间。内核运行时将内核中一些关键的数据结构以文件的方式呈现在/proc目录的一些特点文件中。这些虚拟文件的内容都是动态创建的。

在/proc文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，可将不可见的内核中的数据结构以可视化的方式呈现给内核的开发者，用于调试内核。

在windows系统中，通过=Win32 API的函数调用来完成与内核模式的交互。

将对内核信息的访问交互抽象成了对文件的访问修改，简化了交互过程。

**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。**

对于kseg0的写入操作会缓存到cache中，若通过kseg0访问设备，由于我们需要与设备频繁交互，会不断更新cache中的值，大大降低cache命中率，造成了巨大的时间和性能开销。由于cache机制的特性，设备向映射的内存中写入值得时候，很可能不会刷新cache，导致操作系统读取不到设备传入的值，导致错误。

**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？**

1024\*4KB = 4MB

最大可以指向的其他磁盘块大小和为4MB

支持的单个文件的最大大小为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？**

一个磁盘块有FILE2BLK=16个文件控制块

一个目录最多指向1024个磁盘块，而每个磁盘块下面最多有16个文件控制块

一个目录最多有1024\*16 = 16384个子文件。

**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fs.h文件中的注释

/\* Maximum disk size we can handle (3GB) \*/

#define DISKMAX 0xc0000000

内核支持的最大磁盘大小是3G

**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。**

struct Filefd {

struct Fd f\_fd;

u\_int f\_fileid;

struct File f\_file;

}

struct Fd {

u\_int fd\_dev\_id;

u\_int fd\_offset;

u\_int fd\_omode;

}

从Fd结构体与Filefd结构体的组成来看，Filefd结构体包含了Fd结构体的内容。

在打开一个文件时，该文件描述符被分配了一页的空间。

在serve\_open函数中，设置了文件的各种信息，包括f\_fileid以及struct File

struct Fd \* 型的指针指向的那一页中即存放着本身结构体中所含信息，又紧接着存放着f\_fileid以及struct File。由于C语言中对结构体内存位置分配的规定，将struct Fd\*型的指针转换为struct Filefd \* 型的指针，struct Filefd \* 型的指针指向的那一段内存就能够包含Fileid结构体包含的所有内容。

**Thinking 5.7 请解释File, 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 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];

};

File结构体为文件控制块，定义在include/fs.h

在操作系统内核中用来描述和管理文件，是单纯的内存数据，但每一个File均对应了磁盘上的物理实体，类似env进程控制块与对应进程之间的关系。

f\_name存储文件名

f\_size描述文件的大小，即引用块所占总大小，单位为Byte

f\_type描述文件类型，0为Regular File, 1为Directory

f\_direct[NDIRECT]存储10个直接指针

f\_indirect指向间接磁盘块，存储间接指针

dir指针指向此文件所在的目录文件的文件控制块，仅存在于内存中

f\_pad域是为了填满BY2FILE，起占位作用

struct Fd {

u\_int fd\_dev\_id;

u\_int fd\_offset;

u\_int fd\_omode;

};

Fd结构体定义在user/fd.h

主要在用户进程中使用，存储文件的打开信息,是用户进程对于其要访问的文件的描述。

fd\_dev\_id存储对应文件存储设备的id，在serve\_open函数中设置，

在console.c，以及fd.c的read函数中有用到

fd\_offset存储文件存储在外设中的位置（偏移），在fd.c的write和read函数中使用

fd\_omode存储文件打开的权限位，具体到只读只写还是可读可写。

struct Filefd {

struct Fd f\_fd;

u\_int f\_fileid;

struct File f\_file;

};

相对Fd增加了f\_fileid与file控制块的内容

Filefd结构体是供文件服务系统描述文件时使用的

f\_fileid存储了文件在被打开的时候，被文件服务进程赋予的一个用来标识文件的独一无二的id

f\_file则是对应文件的File结构体

struct Open {

struct File \*o\_file; // mapped descriptor for open file

u\_int o\_fileid; // file id

int o\_mode; // open mode

struct Filefd \*o\_ff; // va of filefd page

};

Open结构体也是供文件服务系统描述文件时使用

主要是为了帮助文件系统同时为多个用户进程提供服务，并全局管理已打开文件

o\_file：对应文件的File结构体指针

o\_fileid：对应文件的id，也就是Filefd中的id域存储的值应该是相同的

o\_mode：表示文件被赋予的权限位

o\_ff：Filefd结构体的虚拟地址

Fileidj结构体是Fd结构体的拓展，包含了Fd中没有但是文件服务进程需要知道的重要信息，这样的设计架构凸显了同一个文件在不同的进程中被突出的部分不完全重叠，是一种良好的设计手段。

**Thinking 5.8 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？**

在serve函数中，先调用了ipc\_recv函数接收其他进程的请求，接下来使用switch语法，对于每种请求，都调用与其相关的serve\_\*函数。

阅读同在serve.c文件中的各个serve\_\*函数，可以看出每一个serve\_\*函数，在执行相应操作后，都会调用ipc\_send函数来通知对方进程。而文件ipc.c中实现的ipc\_send函数来看，它会调用我们之前实现的syscall\_ipc\_can\_send函数，将对方进程设置为runnable状态，并调用syscall\_yield函数来切换进程，故serve函数中的for(;;)循环不会一直处于运行状态，内核也就不会进入panic状态。

**二、实验难点**

**（一）、IDE磁盘驱动**

今年IDE磁盘驱动程序与往年的0S课程相比做出了很大的改变，将对磁盘内容的读写放在了用户态，通过两个syscall系统调用来实现。而阅读网络上的往年代码，往年对这一部分的实现是通过内核态下的汇编函数，用lw、sw指令直接对寄存器进行操作。

所以磁盘驱动的书写就按照指导书中给出的汇编代码，使用C语言对其进行翻译。其中需要注意的是bcopy在使用时，对1位与4位数据，即按字拷贝与按字节拷贝的处理。

**（二）文件系统结构**

这一部分中关于磁盘文件系统布局的内容比较容易理解，与之前管理内存资源和进程控制块不同的是，对于磁盘资源的管理采用了位图来表示每一个Block是否被使用。

我觉得这次实验最难的部分，是fsformat.c这个对磁盘操作的C语言程序，在课下的测试中，我们正是运行了这个程序创建出一个名为fs.img的文件供内核使用。

（1）

在fsformat.c的main函数中，先调用init\_disk来初始化磁盘，再调用write\_file向dirf文件下写入路径path指向的文件。

write\_file中先调用我们自己实现的函数create\_file在dirf文件下新建一个文件索引结构体，即\*target，然后使用C语言库函数strrchr从path中截取出文件名，拷贝到target结构体中。使用open打开这个文件。

（2）

接下来就是一些列对文件内容的操作：

lseek(fd, 0, SEEK\_END)获取文件大小

lseek(fd, 0, SEEK\_SET)

while循环中，调用read读入文件，再使用sava\_block\_link将这个块连接到新创建的文件中。

整个文件读取结束后，使用close()函数关闭文件。

（3）

flush\_bitmap将nextbno之前的所有块都标记为不可用。

（4）

前面的一系列操作都是在内存中完成，将文件结构完善之后，才将内存中的文件内容写入到文件中。

调用finish\_fs函数

先用memcpy拷贝超级块内容，再打开文件，对磁盘文件中的1024个block，都进行大小端转换，再写入文件中，最后关闭文件。

上面所述的4步内容，完成了对一个文件的创建，在熟悉其流程的过程中，我们也掌握了文件系统结构的主要知识。

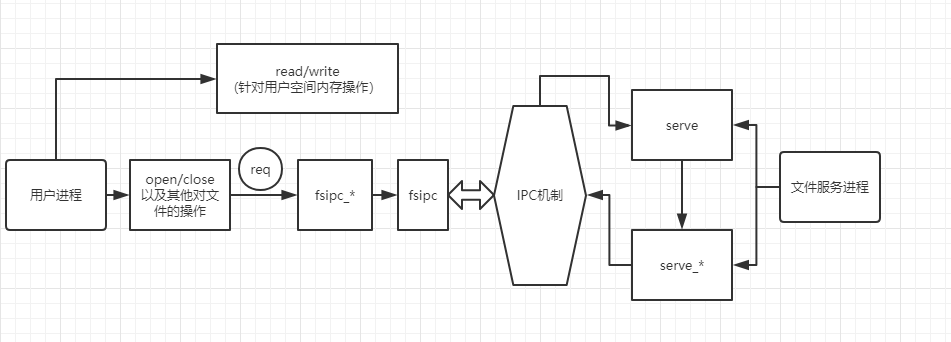
**（三）文件系统的用户接口**

这一部分的内容也令人头疼。我们操纵系统中的文件服务，是一个独立的用户进程，通过IPC的形式供其他进程调用，进行文件读写操作。所以我们在init.c中，在内核运行时，通过ENV\_CREATE(fs\_serv)启动文件系统服务进程。而用户进程在对文件进行操作时，使用ipc\_send/ipc\_recv与fs\_serv进程交互，完成文件的增删改查操作。

除去与内存进行交互的read/write函数之外(主要靠user\_bcopy函数来实现),要想实现对文件的访问，必须通过open/close函数（前面的fsformat也用到过这个）。

open/close函数，调用了fsipc\_open/close函数，在这些函数中，对请求使用Fsreq\_\*的结构体进程封装，并调用fsipc函数，通过ipc\_send与ipc\_recv机制与文件系统服务进程交互。

文件系统服务进程（主要实现在serc.c）中，则对传入的req进行分析，并执行相应的serv\_\*函数，进行对文件的操作，再通过IPC机制向用户进程返回消息。

文件系统进与用户进程交互的示意图如下:

**三、体会与感想**

在写Lab5代码的过程中，并没有遇到那种特别难发现的bug，主要是先前对于ipc\_can\_send函数的书写不完全，导致了lab5卡了几天，后来在这个系统调用中增加了页共享的内容解决了这个问题。

在阅读fs.c、fsformat.c、file.c这几份文件时遇到了很大的困难，由于牵扯内容很多，一开始甚至连什么是属于文件服务进程的内容，什么是用户进程的内容都分不太清楚。过完这一部分的内容，需要一段连续的较长的时间，一旦中断，貌似就得从头看起，不知道前面都实现了什么。

在花了大量时间之后，才对文件系统服务的大体流程有了一定的了解，orz……

**四、指导书建议**

lab4与lab5，关于sys\_ipc\_can\_send函数的衔接，可以描述得更清楚一些，lab4不需要共享页面，lab5需要通过页面完成进程间通信。

对于create\_file函数的注释可以更多一点……