Lab5 实验报告

19373135 田旗舰

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 文件系统是一种特殊的、由软件创建的文件系统，/proc 文件系统只存在内存当中，而不占用外存空间。

最初开发/proc 文件系统是为了提供有关系统中进程的信息。但是由于这个文件系统非常有用，因此内核中的很多元素也开始使用它来报告信息，或启用动态运行时配置。内核使用它向外界导出信息，/proc 下面的每个文件都绑定一个内核函数，用户读取其中的文件时，该函数动态地生成文件的内容。

Windows 操作系统将 I/O 请求转化为相应的 IRP 数据，进行内核与用户的通信。

这样设计的好处是将内核与用户的交互抽象为了文件，简化了交互过程。

可以对组织进一步优化，例如 sysfs 是一个与 /proc 类似的文件系统，但是它的组织更好。

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。

内核在 kseg0 中，如果通过 kseg0 访问设备会导致错误地读写内核的数据。

不同种类的设备由于地址不同，设备寄存器的映射也不同，因此操作会有差异。

Thinking 5.3

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？

一个磁盘块最多存储 1024 个指向其他磁盘块的指针，每个磁盘块大小为 4KB，因此我们的文件系统支持的单个文件的最大大小为 4KB * 1024 = 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？一个目录下最多能有多少个文件？

根据 BY2BLK 为 4096 和 BY2FILE 为 256 可知，一个磁盘块中最多能存储 16 个文件控制块，一个目录最多指向 1024 个磁盘块，因此一个目录下最多能有 1024 * 16 = 16384 个文件。

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 mmu.h 的内存布局图，0x7f3fd000 以上的用户空间都已经被使用，因此最多到达 0x7f3fd000，所以支持的最大磁盘大小为 0x7f3fd000 - 0x10000000 = 0x6f3fd000。

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，因为已经到达了内核空间，可能会覆盖内核数据，从而使整个操作系统无法正常工作。

Thinking 5.7

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。

文件描述符被映射到了 FILEBASE，即 0x60000000 之下的 4M 空间，每个文件描述符占 1 页（4KB）；打开的文件被映射到了 0x60000000 之上的空间，每个文件占 4MB。

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。

```
36 // file descriptor
37 struct Fd {
38         u_int fd_dev_id;
39         u_int fd_offset;
40         u_int fd_omode;
41 };
42
43 // State
44 struct Stat {
45         char st_name[MAXNAMELEN];
46         u_int st_size;
47         u_int st_isdir;
48         struct Dev *st_dev;
49 };
50
51 // file descriptor + file
52 struct Filefd {
53         struct Fd f_fd;
54         u_int f_fileid;
55         struct File f_file;
56 };
```

因为在结构体 Filefd 中储存的第一个元素就是 struct Fd，因此对于指向同一个 struct Fd 的 struct Fd* 型的指针 和 struct Filefd* 型的指针，他们的指针实际上指向了相同的虚拟地址，所以可以通过指针转化访问 struct Filefd 中的其他元素。

Thinking 5.9

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。

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 表示读写文件时距离文件头的偏移量，fd_omode 表示文件打开的读写模式，Fd 结构体并不对应磁盘上的物理实体，是为了方便对打开的文件进行记录和读写管理的单纯的内存数据。

Filefd 表示文件描述符和文件的组合，f_id 为文件描述符，f_file 表示文件的 id，f_file 为文件控制块，其中文件描述符为单纯的内存数据，文件的 id 及文件控制块对应磁盘上的物理实体。

Fd 和 Filefd 结构体均在 user/fd.c 中有所使用。

```
10
11 struct Open {
12         struct File *o_file;    // mapped descriptor for open file
13         u_int o_fileid;                 // file id
14         int o_mode;                             // open mode
15         struct Filefd *o_ff;    // va of filefd page
16 };
```

Open 结构体在文件系统服务进程中用于存储打开的文件的相关信息，o_file 指向对应的文件控制块，o_fileid 表示文件 id，用于在 opentab 中查找对应的 Open 结构体，o_omode 记录了文件打开的模式，o_ff 指向对应的 Filefd 结构体。

Open 在 fs/serv.c 中有所使用。

Thinking 5.10

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。

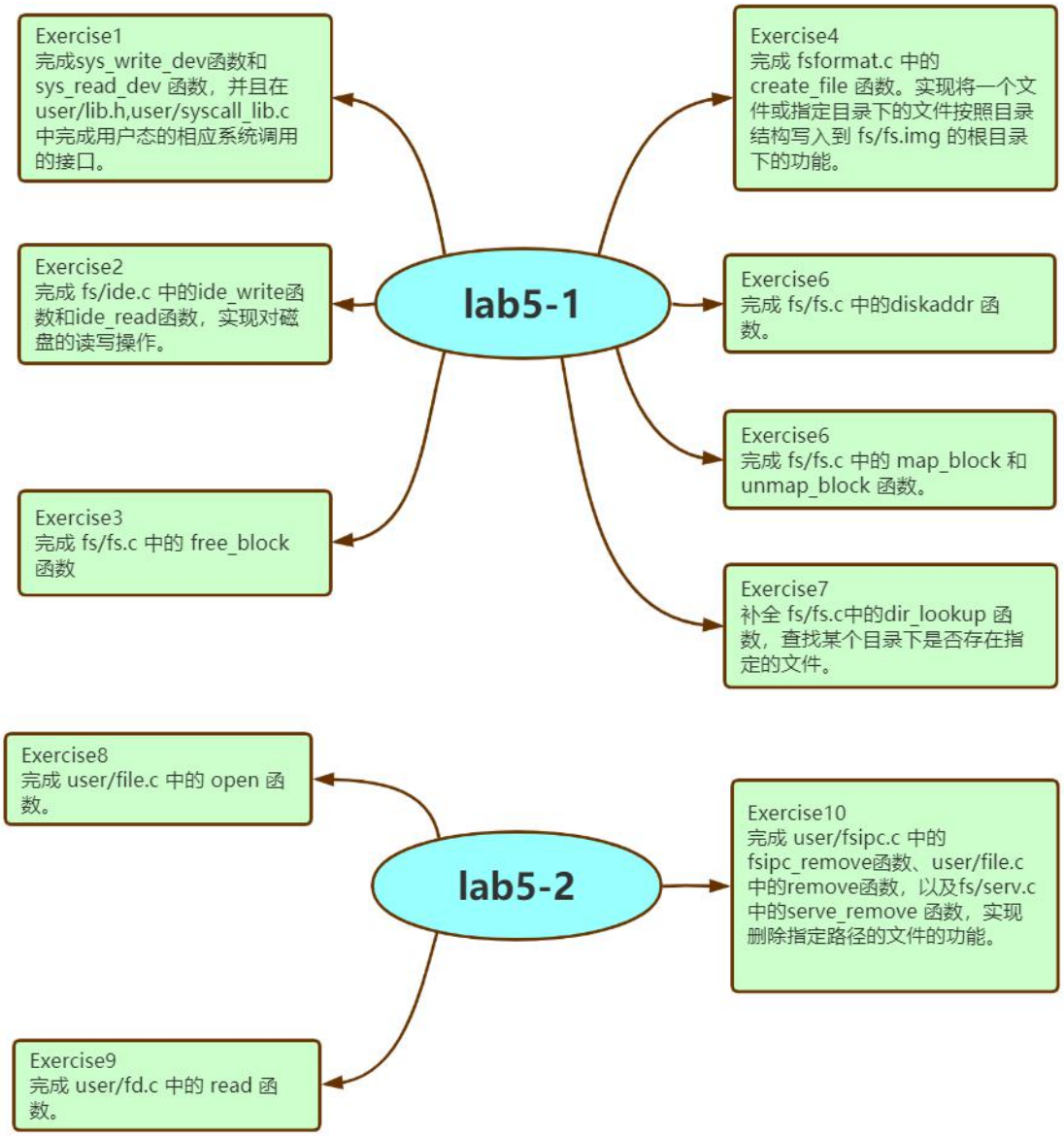
因为在执行 ipc_recv()时进程的状态被设置为了阻塞态，从而不会被 CPU 调度，需要其他进程进行通信才会继续执行，不会一直执行循环，因此不会导致整个内核进入 panic 状态

Thinking 5.11

观察 user/fd.h 中结构体 Dev 及其调用方式，综合此次实验的全部代码，思考这样定义和使用有什么好处。

采用了函数指针，这样可以不仅更灵活地定义各个函数，而且可以在不同文件中只定义需要的函数，更符合微内核的设计。

二、实验难点图示



与 lab3 和 lab4 相似，我认为本次实验的难点仍然不在 exercise 的函数填写本身，而是在于理解整个文件系统，包括每个函数、每个文件的功能，不同函数、不同文件之间的调用关系等。因此，需要我们去阅读大量的代码，而刚开始阅读时往往非常困难，常常遇到函数中有多个不同文件函数的嵌套调用，如果不保持头脑清醒，常常会在读完了一个函数定义后忘记了自己是从哪个地方看到了这个函数的调用，因此，理清函数之间的调用关系并不容易，需要反复阅读实验代码，建立起整个文件系统的框架。此外，也要分清哪些工作是由操

作系统完成，哪些工作是由 CPU 完成，例如 PTE_D 的设置即由 CPU 完成，操作系统只负责检查和清除。

对于本次实验的 exercise，我认为具体的难点如下：

1.用户态磁盘驱动程序的编写

表 6.2: Gxemul IDE disk I/O 寄存器映射

Offset	Effect
0x0000	Write: Set the offset (in bytes) from the beginning of the disk image. This offset will be used for the next read/write operation.
0x0008	Write: Set the high 32 bits of the offset (in bytes). (*)
0x0010	Write: Select the IDE ID to be used in the next read/write operation.
0x0020	Write: Start a read or write operation. (Writing 0 means a Read operation, a 1 means a Write operation.)
0x0030	Read: Get status of the last operation. (Status 0 means failure, non-zero means success.)
0x4000-0x41ff	Read/Write: 512 bytes data buffer.

需要理解 gxemul 设备寄存器映射表，不能简单地直接调用系统调用，并且要区分读写的区别，读操作需要先写入读操作，再从缓冲区中读取数据，而写操作则需要先将数据写入缓冲区，再写入写操作。

2.fsformat.c 中的 create_file()函数

函数本身并不难，但是需要理解 fsformat 是如何来创建一个磁盘镜像文件的。

3.理解 fs/serv.c 文件系统服务进程，通过进程间通信进行文件操作。

三、体会与感想

本次实验相比于 lab3 和 lab4 难度有所降低，也可能是因为有了前面的铺垫，理解起来也相对容易了一些。我在 lab5 上花费的时间大约为 10 个小时，其中 5 个小时用来完成函数的填写，5 个小时用来阅读代码并进行理解，尽管还有一些细节不能彻底说清楚，但已经能够理解整个文件系统的框架。

临近课程的尾声，我对于操作系统的理解也由简单—复杂—再到简单，渐渐感觉到操作系统复杂但又精巧而简单的设计。

四、指导书反馈

部分官方代码有误,例如 fsformat.c 中 disk_init()函数,memset 语句的大小的参数似乎有些无法理解，个人感觉应该修改为：

```
119     for(i = 0; i < nbitblock; ++i) {
120 //wrong!!! why is NBLOCK/8, it should be BY2BLK
121         memset(disk[2+i].data, 0xff, BY2BLK);
122     }
123     if(NBLOCK != nbitblock * BIT2BLK) {
124 //wrong!!! why is BY2BLK, it should be BIT2BLK, look at step2 this opera
125         diff = NBLOCK % BIT2BLK / 8;
126         memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
127     }
128
```

五、残留难点

fs/Makefile 中对于 fsformat 的编译过程还没有理解清楚。