Lab5实验报告

扶星辰 19377251

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？

A：/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。

在windows系统中，通过Win32 API函数调用来完成与内核的交互。

proc文件系统的设计将对内核信息的访问交互抽象为对文件的访问修改，简化了交互过程。具有完整特征的/proc入口项可以相当复杂。

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。

A：kseg0是存储内核的区域，若将对设备的写入也缓存到Cache中，可能会导致读写内核区域出错。

Cache只有替换时才会将更改写入设备，可能会导致在一段时间中对外设没有输出。

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。

A：MOS操作系统使用文件控制块（File 结构体）来描述和管理文件。对于普通的文件，文件控制块其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录所包含的文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。

Unix/Linux 操作系统中的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"。

Unix/Linux系统允许，多个文件名指向同一个inode号码。

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？

A：1个磁盘块中最多能存储16个文件控制块，一个目录下最多能有1024*16=16384个文件，单个文件的最大大小为 4KB*1024=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？

A：DISKMAX = 0x40000000，1GB。

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？

A：不能，可能在运行中覆盖掉内核区域的内容。

```
#define DISKNO 1
IDE磁盘编号

#define BY2SECT 512
每一个扇区有多少字节

#define SECT2BLK (BY2BLK/BY2SECT)
BY2BLK:一个磁盘块大小，所以SECT2BLK是一个磁盘块里包含几个扇区

#define DISKMAP 0x10000000
在 Gxemul 中，console设备被映射到 0x10000000

#define DISKMAX 0x40000000
可装载的最大磁盘大小
```

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。

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。

A：C语言的指针都是相同的结构，只要地址是有效的，便能相互转换。

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。

A：显然会共享

```
int fd = open("./test.cap",O_RDWR);
int pid = fork();
if (pid == 0) {
    printf("child\n");
    printf("%d\n",fd);
    printf("%d\n",lseek(fd,3,SEEK_CUR));
}else {
    printf("father\n");
    printf("%d\n",fd);
    printf("%d\n",lseek(fd,0,SEEK_CUR));
}
```

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。

A：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，以便对文件进行管理。

`Filefd`结构体记录文件详细信息

- `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

Open结构体在文件系统进程中储存文件相关信息

- `o_file`指向了对应的文件控制块
- `o_fileid`表示文件id
- `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。

A：我们的操作系统：

同步：将进程阻塞，直到消息返回再执行。

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?

A：serve函数调用ipc_recv函数，最终调用到sys_ipc_recv函数，我们可以得知在进程准备接受通信时是

会等待的，可见serve中执行到ipc将会处于阻塞状态，不会导致整个内核进入 panic 状态

实验难点图示

本次实验的第一个难点是lab5-1的两个exercise，其中第一题是完成系统调用的读和写，逻辑比较简单，判断地址是否合法然后使用bcopy即可，第二个题目是根据gxemul的磁盘来读取，需要设定特定地址上的值来说明本次磁盘读取的操作类型、操作的目标地址数，总体还好，根据指导书上面的说明一步步写，注意一点就是大小端问题，需要写入4字节的int值0\1。

其次是lab5-2，里面有些函数刚写的时候云里雾里的，例如user/file.c的open函数，一开始无从下手，后面滤清思路后好很多，首先根据path找到对应的文件，然后读内容，最后返回指针即可，后续的5-2-exam是针对open的不同权限位，代码量非常小，需要对这部分有足够深的理解才行。

本次实验填写代码部分个人认为没有lab4难，根据注释以及现有的代码可以轻松的完成，bug部分随着时间的增加，通过测试程序定位bug的能力也越来越好。并且此次实验有补充指导书，很好的梳理了有关.c文件的关系以及定义函数之间的调用关系。

难的部分还是充分阅读和理解写好的代码，并且在课上的时候能够很好的利用已有的代码完成更多的功能。

体会与感想

lab5结束也说明os实验接近尾声了，小小总结一下整个实验的收获，实验跳出了理论的框架，自己手写代码使得对操作系统的理解更加深刻。

指导书反馈

希望指导书的提示可以再多一点（，有些话对初学者还是挺谜语人的。

残留难点

无