Lab5 实验报告

游子诺

2019 年 5 月 25 日

1 实验思考题

Thinking 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 文件系统实际上是一个伪文件系统，存储的是内核运行状态下的一系列文件，通过访问和修改这些文件，能够读取内核实时的运行状态甚至修改内核的运行状态。其文件有以下特点，第一是文件的时间戳基本都是系统的当前的时间，这是因为文件本身的内容时刻存储于内存中，时刻在刷新，第二是文件大小常常是 0 字节，这是因为访问 proc 文件时系统并不会真正地显示文件的内容，而是执行其他的命令显示出当前内核的一些运行状态（访问 proc 文件其实可以被看做是一系列的命令）。

在 Windows 环境下，提供了 windows.h、Zwxxx 等内核函数来实现对内核数据结构的访问。

对于 proc 的好处，其通过虚拟文件系统将用户使用和实现细节抽离开，通过直观的访问文件即可获知内核的相应数据；对于 proc 的改进之处，由于是以文件为单位进行访问，每个文件中都会有大量的内核数据，数据量过多将导致后期的解析工作很麻烦，不属于按需存取，因此应该细化 proc 目录下的文件内容，并可以通过一定的方式增加参数，达到按需存取的目的。

Thinking 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 和 kseg1，硬件在处理时均是将其减去固定的虚拟地址得到实际物理地址进行访问的，差异在于是否经过缓存器。通过 kseg0 访问设备时，在读和写上都可能出现问题：

- 读：当第一次读某个块时，快表将会保存块中的内容，再次访问块中地址将会直接从块里取得数据。这样的缓存逻辑对于内存而言是正确的，但是对于外设每次读取的内容其实可能是不同的，可能随着时间的变化或读取的行为，外设寄存器的内容自动地更新，这样从 tlb 中得到的数据其实是错误的。
- 写：写数据时，其实有“写穿”和“写缓存”两种方式，写穿透的方式写设备没有问题的，但如果在写缓存的情况下和外设交互，那么数据其实并没有写到设备寄存器上，只有当下一次读取这个地址时，tlb 才会将内容写到设备寄存器中，因而出现问题。

Thinking 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？

由于间接指针所指向的数据块的前 10 个位置是空余的（为直接指针留下的无用空间），因此单个文件最大大小用间接指针所指向的数据块所能容纳的数据块指针衡量，一共为 1024 * 4 KB = 4 MB。

Thinking 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？

- 代码中，文件控制块叫做 struct File*，其大小通过定义无关的 pad 数组严格控制为 256 B。
- 因此一个磁盘块最多能够容纳 4 KB / 256 B = 16 个文件控制块。
- 一个目录其实也可以被看做是文件，只是其文件数据是一个一个的文件控制块，由思考题 3 可知，一个目录下最多有 4 MB / 256 B = 16384 个文件。

Thinking 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？

文件系统用自己的虚拟内存空间充当磁盘缓存的位置，起始地址为 $0x1000000$，很显然其最大空间肯定不能达到文件系统进程自身的用户栈 $0x7f3fd000$ 地址，同时通

过对 serv.c 代码的观察，可以看到 struct Open 结构体中的 o_off 存储了从 FILEVA-$0x60000000$ 起始的以页为单位的地址，经过探究其实际用于存储 struct Fd 文件描述符并通过 ipc 返回给用户进程，因此 FILEVA 以上的空间也不能使用，因此文件块缓存的实际空间是 $0x10000000 - 0x60000000$，大小为 1G + 256 MB。

*P.S. 指导书上的块缓存上限到 $0xc0000000$ 显然是错的，都延伸到内核地址空间去了显然不对。

Thinking 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。

仔细观察 fd.c 文件中 INDEX2FD, INDEX2DATA 等一系列转换函数后我们可以发现，每个 struct Fd 结构体实例其实都会被存储在一页的大小空间里（虽然 struct Fd 显然没有一页的大小，非常浪费空间），而这一页的内容并不会通过 mem_alloc 等方法获得，而是文件系统 serv.c 中的 open 函数会借助 ipc 把在其虚拟地址空间中包含 struct Fd 数据的一页映射给用户进程，使得用户也能够访问其中的数据。

有了以页为大小的空间储备下，struct Fd 向更大容量范围的 struct Filefd 转换就变得可行：

- struct Filefd 是 struct Fd 的升级版本：struct Filefd 结构体的第一个成员变量就是 struct Fd，因此强制转换从数据排列上不会出现问题。
- 结构体指针的强制转换是为了获取更强的数据访问权限：在 struct Fd 所在的那一页内存中，实际上 serv.c 已经按照 struct Filefd 的格式把数据填得满满当当了，如果只用 struct Fd* 指针，那么是没有“权限”（从 C 语言语法的角度上来说）访问 struct Filefd 中额外的成员变量的。
- * 要完成这道思考题需要清晰地知道：struct Fd 结构体在虚拟内存中的存储特点；serv.c 的 open 方法是实质填充 struct Fd 和 Filefd 的函数（填充好以后映射到用户进程里去了）；结构体指针访问结构体数据时编译器的实现本质。

Thinking 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。

1. File 结构体
    - 类型：实体数据

- 存在位置：硬盘、文件系统虚拟地址的块缓存（按需载入了以后）、用户进程虚拟地址中的文件 open 空间（open 以后）。控制块本身在硬盘上以”文件“数据的方式存在数据块中，只是这时的”文件“不是普通文件而是文件目录。
- 作用：一个文件结构体与一个文件/文件夹一一对应，结构体是一个**地图**——存有文件/文件夹的大小、类型等基本信息和文件/文件夹所对应真实的所在数据块。

2. Fd 和 Filefd 结构体

   - 类型：内存数据
   - 存在位置：文件系统虚拟地址的 FILEVA 段 ($0x60000000$) 和用户进程的虚拟地址的 FDTABLE 段，结构体所在地址均是每一页的起始地址（独占一页），并且在文件系统和用户两个进程中，同一个 fd/filefd 结构体对应的都是同样的物理页。
   - 作用：struct fd 是文件描述符，struct filefd 是文件描述符的增强版，思考题 6 中已经阐明独占页的模式两种结构体指针可以强制转换。这两类结构体服务的对象主要是用户进程，当用户打开一个文件时，其首次向文件系统发起 open 请求后，文件系统通过访问磁盘（直接方式是通过缓存）数据形成其想要文件的 struct fd/filefd 结构体，并把含有结构体的这一页以“映射分享”的方式发送给用户进程，用户拿到其想要文件的文件描述符后，就可以知道文件的基本信息，进而不断向文件系统发送 map 请求把文件系统中硬盘的块缓存数据以”映射分享“的方式发过来，实现 open 后文件数据就在进程自己的虚拟空间下目的。

3. Open 结构体

   - 类型：内存数据
   - 存在位置：文件系统进程
   - 作用：当文件被 open 时，一个 open 结构体建立，用以保存这个被 open 文件的数据，方便用户进行后续对这个文件进行操作。生动型解释请见图示部分。

Thinking 8

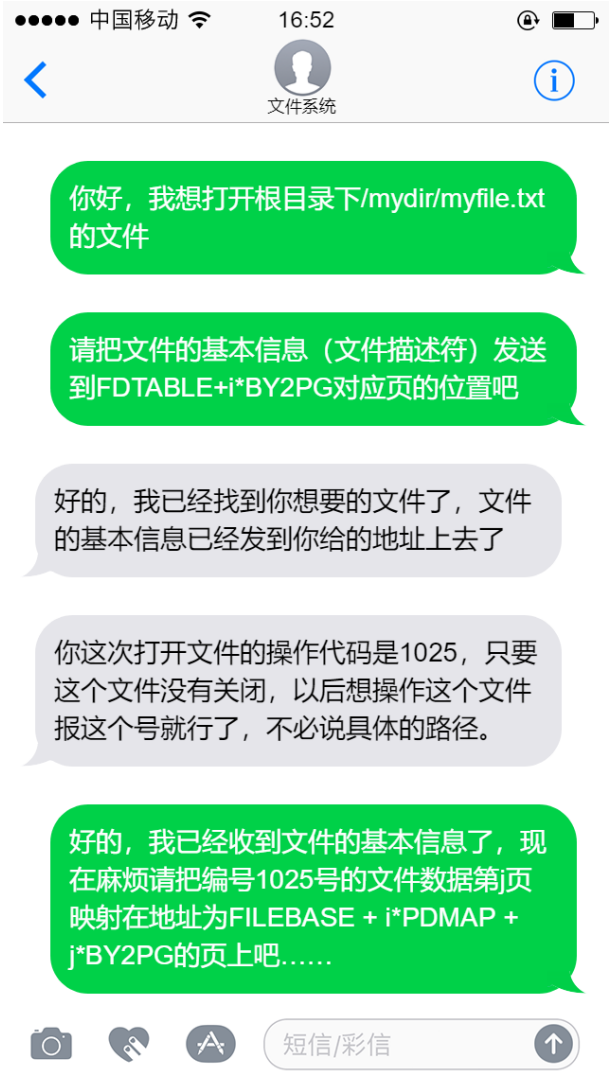
阅读 serve 函数的代码，我们注意到函数中包含了一个死循环 for (;;)...，为什么这段代码不会导致整个内核进入 panic 状态？

- 文件系统是用户进程，会被打断并切换的。

- 因为在 for 循环下面有一个 ipc_recv 函数，这个函数用于接受其他进程向文件系统发送的文件操作请求，如果当前没有请求，文件进程会被阻塞不运行。

2 实验难点图示

在 lab5 中，重点是体会用户进行透过文件系统操作文件的具体过程，在编程练习中我认为对我理解整个流程最大的是 open 函数的完成，如果你有耐心地使用 ctags 等工具逐层递归下去，你就会很清晰地了解到这个流程的运行状态，而其他的操作相比之下都是类似而且更简单的，因而在分析其他操作时会显得更加游刃有余。在这里，我用短信对话的方式展示了 open 环节中用户进程和文件系统进程之间的交互，这个交互是一个中心枢纽，但是仅仅了解其也远远不够，**需要以此为出发点向文件系统和用户进程两方向延伸了解**



其中需要着重注意以下方面内容：

- 用户进程端：open 函数中实质上以一次 open+ 多次 map 的名义向文件系统发送了请求，一次 open 目的是为了取得文件描述符结构体，多次 map 为的是获取文件数据到用户的虚拟空间中。
- 用户与文件系统间的 ipc 交互：ipc 通讯的原理；用户进程向文件系统发送请求时

附带的一页数据里采用的 struct FSREQ_ 类结构体编码；通过指定接收地址的方式接收整页的数据。

- 文件系统端：fs.c 和 serv.c 之间工具箱和工人的关系；serv.c 中国的 open 函数如何申请 open 结构体以及利用 fs.c 中函数得到的数据来填充 open 结构体；文件系统最终向用户进程传递的内存数据何时生成，存在何地？

3 体会与感想

3.1 虚拟页的 alloc 和 map

以往 lab 中，一张物理页对应到不同进程的虚拟页的情况并不多见，而在 lab5 中这种应用多次出现，例如文件系统将硬盘内容块缓存后以 syscall_mem_map 的方式将内容映射到用户进程为打开文件预留的虚拟内存空间，用户进程把请求的详细信息放到固定的一页上以“映射”的方式发送给文件系统，文件系统把一页上的 struct Filefd 填充完毕后以”映射“的方式回传给用户并使用 pageref<=1 判断用户是否已经关闭了某个文件……

我认为这种进程间的通讯方式让 ipc 通讯变得非常巧妙，除了能够获取其他进程传递的数据，还能够：

- 根据传递页标志位的不同还能够共享编辑同一页内容，进而甚至绕过 ipc 方式的通讯。
- 分析虚拟页所对应物理页的引用量判断其他进程的运行状态（如文件是打开还是关闭了）。

总之，lab5 中进程间各种数据通过 ipc 传递的过程令我体验到了使用 IPC 通讯处理复杂数据与操作的巧妙性，也让我感受到了页式管理和虚拟内存管理的巧妙与前瞻性。

3.2 文件系统的用户接口

之前了解到 serv.c 和 fs.c 两份代码的关系是工人与工具的关系，而在 user 文件目录下，我们其实可以看到与文件系统相关的两份代码是 file.c 和 fd.c，这两份代码的关系在我看来有些类似于面向对象中接口以及其实现的关系。

本段所指的文件皆是抽象的文件，不局限于 lab5 实现的 IDE 磁盘文件。一个用户进程操控文件的最重要方法就是文件描述符，对文件的操作常有**读、写、关等**，文件描述符中注册了该“文件”所对应的操作设备是哪个，进而知道在**用户端在向对应操作设备发送指令时应该用哪个具体的实现函数**。当然，在本次 lab 中，file.c 具体实现 fd.c 所定义的接口时，也自己实现了必要的其他函数，例如 open 函数并没有出现在 fd.c 文件中而是 file.c，这是因为以文件路径为参数的 open 并不是所有“文件”都所共用的，lab6 中的 pipe 类文件就不是用 open 产生文件描述符而是用 alloc 产生。这样以 fd.c 为接口的涉及使得文件系统的具体实现对用户几乎是透明的，具体实现（用户端 + 文件系统端）都被接口所覆盖了，使得后期添加各种功能相似或相异的文件系统成为可能（也使得更多的 lab 成为可能

4 指导书残留难点与反馈

4.1 指导书错误反馈

- 5.4.1 节-块的表述：根据 fsformat.c 的实现，硬盘的块都是 4kb 大小的，然而在指导书中有说到存储分区和磁盘超级块（super block）的空间是扇区大小 512 KB，这里的表述是有错误的。

- 5.4.1 节-位图标记算法：指导书中的位图标记算法错误很多，加之指导书前面所讲的块和扇区的概念本来就不明确，因此很容易弄昏同学，建议修改。以下黄色高亮是本段代码中我发现的错误内容：

```
1      nbitblock = (nblock + BIT2BLK - 1) / BIT2BLK;
2      for(i = 0; i < nbitblock; ++i) {
3          memset(disk[2+i].data, 0xff, nblock/8);
4      }
5      if(nblock != nbitblock * BY2BLK) {
6          diff = nblock % BY2BLK / 8;
7          memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
8      }
```

- 5.4.2 节-文件系统详细结构：块缓存的区间是 $0x10000000 - 0xc0000000$，按照虚拟空间 2G+2G 的地址空间划分，很显然这个位置已经直冲云霄到内核区段，甚至超过物理内存了。

4.2 指导书难点

在完成 lab5 的代码时，我耗时最难也是理解最深的部分是 open 函数的填写，但是，在填充这部分代码前，5.5.1 节开头寥寥几句对 open 函数原理的阐释根本无法令我理解如何去完成 open 函数，只有完全搞清楚文件系统这部分的运行机制后回看那部分描述才有所共鸣，因此，这对该部分我的建议如下：

- **增加材料**：增加完成 open 函数前，对文件 open 具体运行机制的讲解，可以加入图示和列表等能够明显看出流程化的表述。

- **强调代码阅读**：lab5 实现的内容不多，但文件系统本身却非常庞大，因此要 lab5 过关必须阅读大量代码对整体处理过程有所把握，open 函数里面使用到了 user/file.c 的 ipc 相关函数，进而涉及了 fs/serv.c 相关函数，不弄清这些代码根本写不出 open 函数。因此，建议指导书在此部分强调阅读文件系统的代码再写函数——**可以思考题的形式指导同学阅读部分文件代码，并回答相关问题。**