

北京大学操作系统实习(实验班)报告

JOS-Lab 5:

File System

黄睿哲 00948265
huangruizhe@pku.edu.cn

June 7, 2012

目录

JOS-Lab 5:.....	1
File System.....	1
内容一：总体概述.....	3
I: File System Preliminaries.....	3
II: The File System.....	3
内容二：任务完成情况.....	4
I: 任务完成列表.....	4
II: 具体 Exercise 完成情况.....	4
1. File System Preliminaries: On-Disk File System Structure.....	4
1.1 Sectors and Blocks.....	5
1.2 Superblocks.....	5
1.3 The Block Bitmap: Managing Free Disk Blocks.....	5
1.4 File Meta-data.....	5
1.5 Directories versus Regular Files.....	6
2. The File System.....	7
2.1 Disk Access.....	7
Exercise 1.....	7
2.2 The Block Cache.....	8
Exercise 2.....	8
2.3 The Block Bitmap.....	12
Exercise 3.....	12
2.4 File Operations.....	13
Exercise 4.....	13
2.5 Client/Server File System Access.....	15
Exercise 5.....	15
Exercise 6.....	18
2.6 Client-Side File Operations.....	19
Exercise 7.....	20
2.7 Spawning Processes.....	21
Exercise 8.....	21
内容三：遇到的困难以及解决方法.....	23
困难一：对文件系统的整个流程理解得不够透彻.....	23
困难二：需要阅读大量代码.....	23
内容四：收获和感想.....	24
内容五：对课程的意见和建议.....	25
内容六：参考文献.....	26

内容一：总体概述

JOS Lab5 实习的主要内容为：实现一个基于磁盘的文件系统。这个文件系统将以一种“微内核”的方式实现（什么是微内核？参看邵老师讲义），文件系统是在用户空间中的一个“文件系统进程”中实现的，而不是在 `kernel` 中，这就大大减少了我们 `kernel` 的复杂性。其他用户进程访问磁盘文件，通过和文件系统进程进行进程间通信 IPC 来完成。

I: File System Preliminaries

这一部分，我们要了解 JOS 文件系统中的基本概念和布局，包括磁盘/文件系统块、超级块、磁盘空闲块位图、文件元数据、一般文件和目录文件等。我们要知道它们在磁盘上都是怎么布局和存放的。

II: The File System

我们将实现 JOS 的文件系统，事实上很多代码都已经给出了，我们只需填补一些关键功能即可。我们采用一种“自底向上”的方式，首先了解 JOS 中对 IDE 磁盘的驱动程序(`fs/ide.c`)，接着到 file system 和 IDE disk 的交界面(`fs/fs.c`)，然后到 file system 本身的功能实现(`fs/fs.c`)，接下来到 file system 和上层 IPC 之间的数据交换和功能调用(`fs/serv.c`, `lib/file.c`, `lib/fd.c` 等)，最后实现一些用户库函数，这些库函数包装 IPC 调用底层文件系统功能。

内容二：任务完成情况

I: 任务完成列表

Exercise	1	2	3	4	5	6	7	8
第一周	Y	Y	Y	Y				
第二周					Y	Y	Y	Y

Challenge	1	2	3	4
第三周		Y		

II: 具体 Exercise 完成情况

1. File System Preliminaries: On-Disk File System Structure

首先明确，我们要做的这个 JOS 文件系统已经比真实的文件系统要简化很多了。它主要提供了这些功能：在一个层次的文件目录结构中进行创建、读取、写入、删除文件。

大多数 Unix 文件系统将磁盘空间划分成两个区域：i 结点区，数据区。Unix 文件系统中给每一个文件分配了一个 i 结点，里面保存了用于描述该文件的元数据(如文件状态信息、指向文件在磁盘的数据块的指针等)。而数据区中则保存了文件的数据块和目录信息。文件目录项中包含文件名和 i 结点指针，如果两个文件目录指向同一个 i 结点，称为硬连接。JOS 文件系统中不支持硬连接，因此甚至连 i 结点都不要了（也没有 i 结点区），我们直接在目录项中记录文件的原数据。

一般来说，文件、目录都会包含多个数据块，这些数据块分散在磁盘之中，也许逻辑相邻的数据块物理上并不一定相邻。而文件系统就是为上层应用隐藏了这些细节的，让它们可以在文件中以任何偏移量读写数据。文件的创建、读写、删除都是在“文件系统进程”内部完成的。

下面是一个很好的类比：

文件系统~磁盘 ⇔ 内存管理~物理内存

它们很相似，不过也会因为不同的考虑、不同的情境、不同的需求而设计得不一样。

1.1 Sectors and Blocks

磁盘读写的粒度，不是以“字节 bytes”而是以“扇区 sectors”为单位，一般一个扇区的大小是 512 字节。而文件系统使用磁盘的粒度，是以“块 blocks”为单位，块的大小当然必须是扇区大小的整数倍。（磁盘：扇区；文件系统：块）

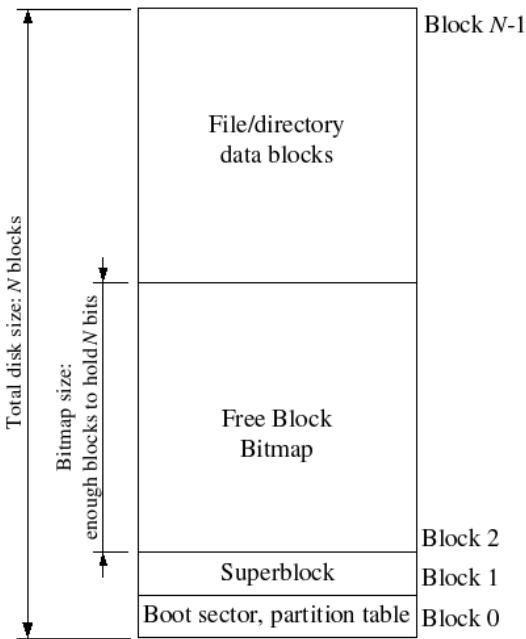
在 JOS 中，我们令“1 块 = 4096 字节 = 8 扇区”，这样文件系统的块大小就和内存管理中的页面大小是一样的。我们要记住这个换算关系，因为磁盘中的扇区、块是按照线性顺序来进行编号的，因此只需要简单的乘除法就能够将“扇区号”和“块号”进行换算。

1.2 Superblocks

文件系统一般会在一些特殊的文件系统块的位置，保存用于描述整个文件系统的元数据的信息，如块大小，磁盘大小，根目录元数据，文件系统装载时间，最后一次检查时间等等。保存这些信息的特殊的文件系统块（对应于相应的磁盘扇区），就成为“超级块 superblocks”。

JOS 文件系统中只有唯一一个超级块，在磁盘的第 1 块（block 1，如图所示）。超级块里面保存了什么信息？参看 inc/fs.h 中 struct Super 结构体的定义得知，里面就保存了一个魔数、文件系统块数、根目录的文件信息。磁盘第 0 块（block 0）中用来干嘛了？在 Lab 1 中我们知道，里面保存了 boot loader 和磁盘分区表，因此超级块 block 1 开始。

在真实的文件系统中，超级块可能有若干块，而且进行冗余存储以保证可靠性。



1.3 The Block Bitmap: Managing Free Disk Blocks

内存管理要记录物理页的使用情况，这是通过一个 page_free_list 链表将空闲页框链接起来实现的。

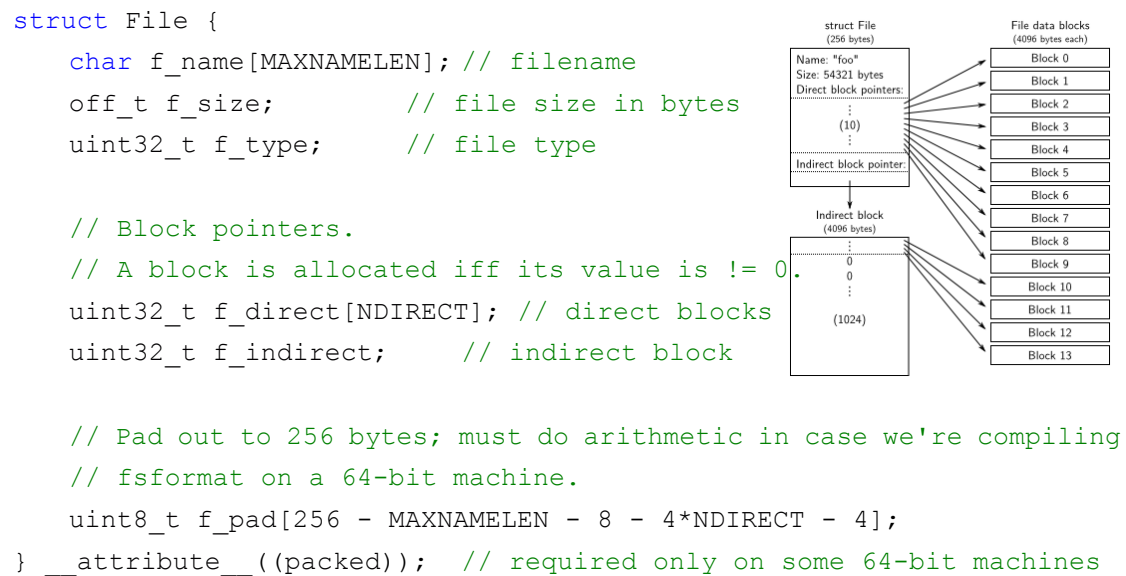
对于文件系统，它是怎么记录磁盘块的使用情况的？它通过位图 bitmap 而不是链表，这是因为磁盘空间一般远大于内存，因此用 bitmap 记录空闲磁盘的情况，显然比链表空间开销来的少。从 bitmap 中找到一个空闲块的代价，要高于从链表中找到一个空闲页框，不过对于文件系统来说，这些开销远远不如 IO 开销来得大。IO 开销主导了文件系统的性能。

为了保存“空闲磁盘块 bitmap”，我们在磁盘块上预留足够大的连续空间，在这个连续空间中，bitmap 中的“位”和“磁盘块”一一按顺序对应。如果一个磁盘块是空闲的，那么相应的“位”应置 1；反之如果已被占用，则置 0。保存 bitmap 的这片连续空间是从 block 2 开始的，紧紧跟在超级块后面，文件系统/磁盘有多大，可以计算出需要多大的 bitmap。

1.4 File Meta-data

如何描述一个文件？这就需要文件的元数据信息，这是通过 inc/fs.h 中 struct File 结构

体来定义的（我们没有使用 i 结点）：



描述一个文件的信息包括：文件名、文件大小、文件类型（一般文件 or 目录文件）；数据块指针（直接块、间接块）。也就这么多，剩下的是为了保证对齐，也即保证一个 File 结构体大小为 256 字节，这样一个 block 就能够正好装下 4 个 File 结构体。

这里谈谈文件的数据块。显然，同一个文件的数据块是逻辑相邻的，但是它们在磁盘中不一定是物理相邻的。因此我们就需要一个数组作为目录，来记录这个文件的第 1 块对应磁盘中的哪个块、第 2 块对应磁盘中的哪个块……这就是一个映射关系，跟我们的页目录/页表一样。

这里还有一个“直接块”和“间接块”的概念：直接块是能在 File 结构体中直接定位到的数据块，直接块只有 NDIRECT 个（其中 NDIRECT=10）；间接块是不能在 File 结构体中直接定位到的数据块，而是需要通过 File 结构体中，指向一个新的磁盘块做为“目录块”，目录块再指向间接块，记录该文件中 NDIRECT 以后的数据块的对应关系。小心，我们这里说的“目录块”是为了索引一个文件的所有数据块而言，而不是文件系统中的文件目录。我们已经知道一个磁盘块大小为 4096 字节，因此任何大于 40KB 的文件必然需要一个“目录块”，而 JFS 支持的最大文件大小应该是 $(10+1024)*4KB = 4136KB \approx 4MB$ 。在真实的文件系统中，对于大文件“目录块”可能还需要拓展到二级、三级。

1.5 Directories versus Regular Files

JFS 中磁盘除了前面几个块拿来保存 boot loader, super block, bitmap 以外，剩下的区域全部都是数据区。数据区中可以保存“一般文件”和“目录文件”。

JFS 中“一般文件”和“目录文件”都用 struct File 来表示，二者通过 type 域区分。文件系统管理一般文件和目录文件的方式完全相同，只是取出任何一块，理解方式不同而已：文件的话就是 byte stream 然后交给用户即可，而目录的话里面就是连续的 File 结构体。

在超级块中，保存了文件系统根目录的 File 结构体，对应一个目录文件，里面是一堆连续的 File 结构体，指向保存在根目录下的文件或下一层的目录。

2. The File System

吹起来容易做起来难。接下来我们将实现 JOS 的文件系统功能。看见只有 8 个 exercise 的时候我开心得有点站不稳了……不过后来我发现，虽然不用写很多，但要读的代码多得我有点站不稳了……而且我还没有读完 T_T。其实开玩笑啦，花两个小时就可以把下面文件中的代码大概过一遍，知道文件里每个函数干什么就行：

- **fs/ide.c** 这里是和硬件打交道的，实现了 IDE 磁盘驱动，知道干了什么就行。
- **fs/bc.c** buffer cache，怎么把数据从硬盘读到物理内存。
- **fs/fs.c** file system，在前两个文件的基础上，实现了文件系统的功能。
- **fs/serv.c** file system server，在文件系统和 IPC 基础上服务于其他进程的 IPC 请求。

注意，这里面包含了文件系统进程的 `umain()` 函数。

- **lib/fd.c** file descriptor，文件描述符，略略看。
- **lib/file.c** 向 file system server 发送 IPC 请求服务。
- **lib/spawn.c** “进程产卵”？！知道它干什么就行，其实里面的代码很多以前都见过了，就是比较长比较麻烦。（连张弛学长都木有仔细看过的代码~~）

如下是需要我们填补的工作：

- (1) 从磁盘中将指定 block 读入内存中的 block cache，或从 block cache 写回磁盘；
- (2) 分配磁盘 block（主要用于创建文件、文件增长等，需要新的磁盘空间）；
- (3) 在“文件中的偏移量”和磁盘 block 之间建立映射；
- (4) 利用 IPC 接口，实现读、写、打开文件的操作。

2.1 Disk Access

我们是通过一个“文件系统进程”来提供文件系统服务。理论上，文件系统进程需要访问文件，而目前我们的 kernel 并没有提供这个服务。在“整体式(monolithic)内核”中，我们可以为 kernel 添加 IDE 磁盘驱动器，然后文件系统进程通过系统调用，来调用内核驱动磁盘的服务。不过在 JOS 中我们并不这么做，我们**直接在用户空间的文件系统进程中实现 IDE 磁盘驱动程序**，然后只需要稍微修改一下 kernel，使得用户的文件系统进程自己具有访问磁盘的特权级别。

为了简化工作，我们在用户空间采用轮询、可编程 IO 的方式实现磁盘访问，而不是中断。当然用中断也可以，不过麻烦在于，类似时钟中断，我们需要在 kernel 中添加相应的设备中断，进行相应的分发，然后在用户程序中进行中断处理。

X86 处理器通过 `eflags` 标志寄存器的 **IOPL** 位来决定保护模式的代码是否可以执行特殊的设备 IO 指令，例如 `IN` 或 `OUT` 指令。因为我们需要访问的 IDE 磁盘寄存器位于 x86 处理器的 IO 空间中而不是被内存映射，因此我们需要做的仅仅是为“文件系统进程”设置相应的权限级别，是使之可以访问这些寄存器，而其他的用户进程不可以。

Exercise 1. `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.

Make sure you can start the file environment without causing a General Protection fault. You

should pass the "fs i/o" test in make grade.

exercise 1 的代码没什么好解释的, 在 kern/env.c 的 env_create() 函数中:

```
// If this is the file server (type == ENV_TYPE_FS) give it I/O privileges.  
// LAB 5: Your code here.  
if (type == ENV_TYPE_FS)  
    e->env_tf.eflags |= FL_IOPL_3;
```

MIT 课程材料中提出一个问题: 我们**如何保证**这个 IO 特权级别的设置 (eflags 寄存器中的 IOPL 位), 在进程间上下文切换时, 是有效而且正确的? 参考张驰学长的报告, **保存现场**是在时钟中断发生时的 Trapframe 里面的 tf_eflags, 而**恢复现场**是在 env_pop_tf() 函数中的 iret 指令。

接下来是要大致阅读一下 fs 文件夹里面的代码.....之前有介绍过了。

2.2 The Block Cache

接下来我们要实现文件系统**“文件缓存 block cache”**功能, 简单地说, 就是在内存管理机制的基础上将磁盘内容读入物理内存的过程。这部分代码是在 fs/bc.c 中。

我们的这个文件系统能够支持的磁盘大小最多只有 3GB。因此, 我们在文件系统进程的线性地址空间中预留了 3GB 的区域来对应磁盘内容, 即 0x10000000(DISKMAP) 到 0xD0000000(DISKMAP+DISKMAX), 这就是一个虚拟内存空间到磁盘空间的一一映射。例如, 磁盘 block 0 映射到 0x10000000 开始的线性地址, 磁盘 block 1 映射到 0x10001000 开始的线性地址, 依此类推。Fs/bc.c 中的 diskaddr() 函数实现了 block 编号到对应线性地址的转换。

应该注意的是, 由于我们的文件系统进程, 不干别的事情, 而且它的线性地址空间和其他进程是相互独立的, 因此预留出 3GB 来映射磁盘是可行的。但是对于真实的系统: 如果是 32 位系统, 那么因为线性地址空间只有 4GB, 而磁盘空间一般比它要大得多, 因此这样映射是不可行的; 不过至于 64 位系统, 这种映射仍然可行。

在 JOS 中, 虽然我们对整个磁盘进行了映射, 但我们显然不会将整个磁盘的内容读入内存; 我们采用一种“按需求读取磁盘”的策略, 也即只有发生了磁盘读取, 我们才会将相应的磁盘内容读入所映射的线性内存地址。什么时候发生磁盘读取? 当然是发生 page fault 的时候了。不过依然要注意, 我们的这个 buffer cache 还是很弱的, 我们不能像真正的文件系统一样维护一个固定大小的文件缓存区, 也就是说维护一个大小固定的 buffer cache, 若超出, 则根据策略选择一些缓存的内存页面写回磁盘。事实上, 我们实现的这个 buffer cache 是**没有上限**的。(稍后的 challenge 中我们将尝试给 buffer cache 增加回收策略, 实现有限的缓冲区。)

Exercise 2. Implement the bc_pgfault and flush_block functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one you wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) addr may not be aligned to a block boundary and (2) ide_read operates in sectors, not blocks.

The flush_block function should write a block out to disk if necessary. flush_block shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's

not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the PTE_D "dirty" bit is set in the vpt entry. (The PTE_D bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, flush_block should clear the PTE_D bit using sys_page_map.

Use make grade to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

exercise 2 代码如下，先贴代码再解释：

```
static void
bc_pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
    int r;

    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("page fault in FS: eip %08x, va %08x, err %04x",
              utf->utf_eip, addr, utf->utf_err);

    if (super && blockno >= super->s_nblocks)
        panic("reading non-existent block %08x\n", blockno);

    // LAB 5: Your code here
    addr = ROUNDDOWN (addr, PGSIZE);
    if ((r = sys_page_alloc (0, addr, PTE_U|PTE_W|PTE_P)) < 0)
        panic("fs/bc.c/bc_pgfault(): page allocation error: %e", r);

    ide_read(blockno*BLKSECTS, addr, BLKSECTS);

    // Check that the block we read was allocated. (exercise for
    // the reader: why do we do this *after* reading the block
    // in?)
    if (bitmap && block_is_free(blockno))
        panic("reading free block %08x\n", blockno);
}
```

这个 bc_pgfault() 就是 page fault 的中断处理程序，原理和 lab 3 中的那个 pgfault() 函数差不多。我们首先将导致 page fault 的线性地址 va 转换成磁盘 block 编号，然后在经过一系列的检查之后，调用 IDE 磁盘驱动的 ide_read() 函数即可。

代码中提出一个问题：**为什么**是在 ide_read() 之后，而不是在之前，用 bitmap 检查读入的 block 是否为空块？答案是：(1) 因为 ide_read() 只负责从磁盘的相应 block 读入内存，这是任何时候都可以进行的操作，因此需要检查读入的块是否为空，也就意味着这一磁盘块

中的内容是否有效。(2)至于 `ide_read()` 为什么在它之后而不是之前检查 `bitmap` 呢？我们注意到 `if` 的条件里面检查了 `bitmap` 是否为空，因此答案就很明显了，这是为了保证在访问 `bitmap` 时产生 `page fault` 能够正确处理（从磁盘 `block 2` 以及后续的 `blocks` 读入数据），而不产生错误。

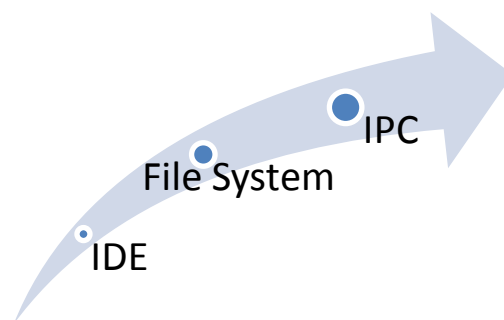
```
void
flush_block(void *addr)
{
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

    if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
        panic("flush_block of bad va %08x", addr);

    // LAB 5: Your code here.
    int r;
    addr = ROUNDDOWN(addr, PGSIZE);
    if (va_is_mapped(addr) && va_is_dirty(addr)) {
        ide_write(blockno*BLKSECTS, addr, BLKSECTS);
        if ((r = sys_page_map(0, addr, 0, addr, PTE_SYSCALL)) < 0)
            panic("fs/bc.c/flush_block(): page mapping error: %e", r);
    }
}
```

这个 `flush_block()` 函数，是负责将 `buffer cache` 中的数据写到对应的磁盘块中。这里我们需要了解“脏页”机制，也就是如果一个可写的页面在从磁盘读入内存后被修改了，那么这一页就“脏了”，这是由 `MMU` 硬件提供相应的机制，自动在页表项添加 `PTE_D` 标志来标记的。因此在 `flush_block()` 函数中，我们在判断完地址有效和是脏页后，就可已调用 `ide_write()` 函数来写回磁盘，同时调用 `sys_page_map()` 函数将页面重新设置为“不是脏页”。

我们实现 `JOS` 文件系统有点“自底向上”的意味，刚刚我们通过驱动程序，实现了文件系统和 `IDE` 磁盘之间的数据交换功能，之后我们还要接着完善文件系统，最后实现上层的 `IPC` 机制和文件系统数据交换的功能。如图所示。



Challenge. The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the `PTE_A` "accessed" bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

尽管我们的 **buffer cache** 已经实现“按需读取”，但是它并没有“回收策略”，也就是说当 **buffer cache** 中的文件块数较多时，它并不能将那些可能不再会被使用的文件块清理出内存。因此，在这个 **challenge** 中我们将尝试实现有限的 **buffer cache**。

题目已经提示我们，可以利用页表中的 **PTE_A** 位，查看这一页是否被访问。**PTE_A** 是由硬件自动设置的，这样我们就不需要手工在每个访问内存的位置设置 **PTE_A** 为“访问过”了。对于“脏”的 **block**，我们调用刚刚写好的 **flush_block()** 函数写回磁盘就好。

联系操作系统原理课上“工作集”的概念，在这里，我们也可以简单实现类似的机制。我们只需要在 **buffer cache** 即将读入一块新的磁盘块之前，检查当前的 **buffer cache** 中保存的磁盘块数和使用情况，然后将“最可能不再被使用”的块释放。

因此，我们可以在 **fs/bc.c** 的 **bc_pgfault()** 函数中，即将调用 **sys_page_alloc()** 分配新的页框映射磁盘块之前，添加一个函数调用：

```
eviction_policier(blockno);
```

eviction_policier() 这个函数就实现了我们的“**buffer cache** 回收策略”。这个函数我写得代码较长，在此给出不完整的代码，重在体现回收的策略：

```
static void
eviction_policier(uint32_t blockno){
    // if the block cache is not full
    if (bc_count < MAX_CACHE){
        bc_count ++;
        blockno_queue[head] = blockno;
        timestamp_queue[head] = timestamp ++;
        head = (head+1) % MAX_CACHE;
        return;
    }

    //Now we choose a cached block to write back, before reading new block.
    for (i = 0; i < MAX_CACHE; i ++){
        addr = (void *) (blockno_queue[i]*BLKSIZE + DISKMAP);
        if ((vpt[PGNUM(addr)] & PTE_A) != 0)    // if accessed
            sys_page_map(0, addr, 0, addr, PTE_SYSCALL)
        else{
            found = i;
            break;
        }
    }

    // If no block is not-accessed,
    // then choose the earliest to be replaced
    if (found < 0){
        int earliest = timestamp_queue[0];
        found = 0;
        for (i = 0; i < MAX_CACHE; i ++){
            if (timestamp_queue[i] < earliest){
                earliest = timestamp_queue[i];
            }
        }
    }
}
```

```

        found = i;
    }
}
addr = (void *) (blockno_queue[found]*BLKSIZE + DISKMAP);
}

flush_block(addr);
if((r = sys_page_unmap(0, addr)) < 0) // free the found block
    panic("fs/bc.c/eviction_policier(): page ummap error: %e", r);
blockno_queue[found] = blockno; // new block replace the found block
timestamp_queue[found] = timestamp ++;

return;
}

```

回收策略如下：

- (1) buffer cache 中最多能够同时容纳 MAX_CACHE 个磁盘 block;
- (2) 分配每个磁盘块时，其编号放入 blockno_queue 数组，同时在 timestamp_queue 数组中记录其时间戳（时间戳是一个静态变量，每次自增 1）；
- (3) 如果 buffer cache 已满，需要选择一个 block 被替换出去；
- (4) 如何选择被替换的 block? 遍历 blockno_queue，查看每个 block 相应虚拟地址的 PTE_A 位，若 PTE_A 显示“未访问”，则选择该 block；同时注意在遍历过程中，将所有 block 相应 PTE_A 位置为 0；
- (5) 若每一 block 均为“访问过”，则通过 time stamp 选择一块最早的 block。

2.3 The Block Bitmap

现在我们从“文件系统和 IDE 磁盘交接面”上升到了“文件系统本身”。

在 fs_init() 函数中初始化了 bitmap，也就是“空闲磁盘块位图”，这里的每一位一一对应着一个磁盘 block。要了解这个机制的实现框架，我们可以参考 fs/fs.c 中的 block_is_free() 函数。

接下来我们要实现 fs/fs.c 中的 alloc_block() 函数，这个函数在 bitmap 中找到第一个空闲的磁盘 block，然后返回它的 block 编号。当我们分配了一个 block，应该马上调用 flush_block() 函数将 bitmap 对应的 block 2 以及后续 blocks 写回磁盘。

Exercise 3. Use free_block as a model to implement alloc_block, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with flush_block, to help file system consistency.

Use make grade to test your code. Your code should now pass "alloc_block".

枚举 block 编号即可，fs/fs.c 中的 alloc_block() 函数代码如下：

```

int
alloc_block(void)

```

```

{
    uint32_t blockno;
    for(blockno = 0; blockno < super->s_nblocks; blockno++)
        if (block_is_free(blockno)) {
            bitmap[blockno/32] &= ~(1<<(blockno%32));
            flush_block (bitmap);
            return blockno;
        }
    return -E_NO_DISK;
}

```

2.4 File Operations

前面已经说过了，fs/fs.c 中实现了文件系统的功能，也就是说里面提供了各种函数，来理解和管理 File 结构体，扫描目录文件，根据绝对路径找到相应的文件等。简单了解一些 fs/fs.c 中每个函数的功能还是很有帮助的。

Exercise 4. Implement file_block_walk and file_get_block. file_block_walk maps from a block offset within a file to the pointer for that block in the struct File or the indirect block, very much like what pgdir_walk did for page tables. file_get_block goes one step further and maps to the actual disk block, allocating a new one if necessary.

Use make grade to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file_rewrite".

接下来我们要实现 fs/fs.c 中的 file_block_walk() 和 file_get_block() 两个函数，它们在文件系统中的地位至关重要。MIT 材料中将这两个函数比喻为 “the workhorses of the file system”。例如，file_read() 和 file_write() 这些相对高层和抽象的函数，都需要建立在 file_get_block() 之上，来将物理不连续的磁盘块中的内容，逻辑上组装起来，读入连续的 buffer 中。

file_block_walk() 函数，非常类似于我们在内存管理中写过的 pgdir_walk() 函数，它的功能是：给定一个文件中的 block 序号，返回这个 block 序号对应的磁盘 block 编号 (Given: block number in this file → Return: block number on the disk)。理解起来比较绕，举个例子更清楚：一个文件是由很多数据 block 组成，互相之间具有逻辑的顺序关系，但物理上不一定相邻。现在我要读这个文件的第 8 个数据 block，于是我就要找到 “直接块” 的第 8 项，看看它对应应在磁盘上的 block 的编号是多少；如果我要读这个文件的第 22 个数据 block，于是我就要找到 “间接目录块” 的第 22-10=12 项，看看它对应应在磁盘上的 block 的编号是多少。理解清楚后，代码如下：

```

static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno,
bool alloc)
{
    int r;
    if (filebno >= NDIRECT + NINDIRECT)

```

```

        return -E_INVALID;

    if (filebno < NDIRECT) {
        if (ppdiskbno != NULL)
            *ppdiskbno = &(f->f_direct[filebno]);
        return 0;
    }

    // A block is allocated iff its value is != 0.
    if (f->f_indirect == 0) {
        if (!alloc)
            return -E_NOT_FOUND;
        if ((r = alloc_block()) < 0)
            return -E_NO_DISK;
        f->f_indirect = r;
        memset (diskaddr(r), 0, BLKSIZE);
        flush_block(diskaddr(r));
    }

    if (ppdiskbno != NULL)
        *ppdiskbno = (uint32_t *)diskaddr(f->f_indirect) + (filebno -
NDIRECT);

    return 0;
}

```

`file_get_block()`这个函数，是对刚刚写完的 `file_block_walk()`函数的包装。它的作用是，给定文件中的数据 `block` 序号，调用 `file_block_walk()`获得相应磁盘上的 `block` 编号，当然如果该数据 `block` 对应的磁盘 `block` 尚不存在，那么就调用 `alloc_block()`分配一个，然后将 `blk` 设置为刚刚分配的磁盘块在内存 `buffer cache` 中对应的线性地址。反正总的来说就是，给定文件中的数据 `block` 序号，找到对应磁盘块，返回相应线性地址。代码如下：

```

int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    int r;
    uint32_t *ppdiskbno;

    if ((r = file_block_walk(f, filebno, &ppdiskbno, 1)) < 0)
        return r;

    if (*ppdiskbno == 0) {
        if ((r = alloc_block()) < 0)
            return -E_NO_DISK;
        *ppdiskbno = r;
        memset (diskaddr(r), 0, BLKSIZE);
    }
}

```

```
        flush_block(diskaddr(r));
    }
    *blk = diskaddr(*pdiskbno);

    return 0;
}
```

2.5 Client/Server File System Access

到此为止,我们已经实现了 **file system** 的功能,也就是用户空间的文件系统进程。现在,我们要继续“自底向上”实现“IPC 调用 **file system**”的界面,这样其他用户进程就可以访问文件系统进程了。

由于其他用户进程不能直接调用文件系统进程的函数,因此,我们通过一种“远程过程调用 (**remote procedure call, RPC**)”的机制,将文件系统进程的功能调用,暴露给其他用户进程。而这个机制就是需要建立在 Lab 4 实现的 IPC (进程间通信) 机制的基础上的。

关于“**远程过程调用 (remote procedure call, RPC)**”的机制,参看张弛学长的 Lab 5 报告。我的理解,RPC 就是用户程序 1 调用一个函数 f1, f1 中**包装**了另一个用户程序 2 的函数 f2 以及相应实现进程通信的 IPC 的接口,这样一来用户程序 1 以为 f2 的功能是在本地实现的,而实际上 f2 是远程的用户程序 2 的函数。

记住现在我们上升到“IPC 调用 **file system**”的界面了,因此文件也相应为 fs/serv.c (IPC 的服务器端), lib/file.c (IPC 的客户端)。

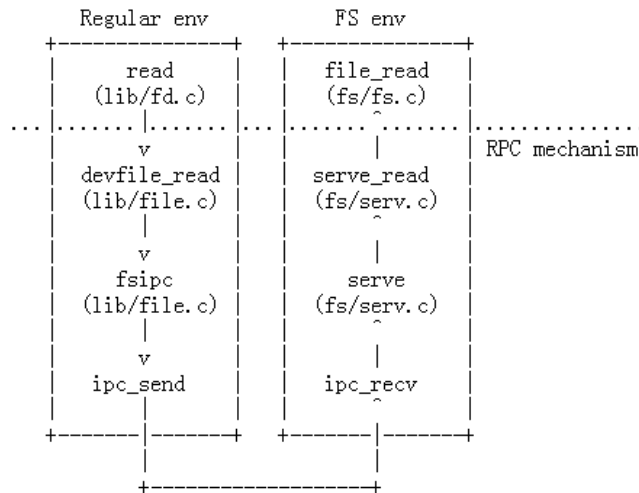
Exercise 5. Implement `serve_read` in `fs/serv.c` and `devfile_read` in `lib/file.c`.

`serve_read`'s heavy lifting will be done by the already-implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.

Likewise, `devfile_read` should pack its arguments into `fsipcbuf` for `serve_read`, call `fsipc`, and handle the result.

Use `make grade` to test your code. Your code should pass "`lib/file.c`" and "`file_read`".

要完成这个 exercise,首先要了解: **我们的数据是怎么样从 file system 进入 IPC 服务器端,再进入 IPC 客户端的?** (在前面的 exercises, 我们已经知道了数据是怎么从 IDE 磁盘进入 **file system** 的了。)下图显示了当某个用户进程向文件系统服务器请求“read”时的控制信号和数据的传递过程:



虚线表示 RPC 机制，说白了，它就是将虚线下方的各种复杂过程包装起来而已。从左上角开始看：

①普通用户进程对“文件描述符”发起一个读操作。

②这个“文件描述符”将这个读操作分发到相应的“设备读”函数，这里是 `devfile_read()`，即“设备文件读函数”（在以后的 lab 中，我们将会有更多的设备，如 pipe，network socket 等，现在的设备是 IDE 磁盘）。

③`devfile_read` 实现了客户端从磁盘上读取文件的功能，注意到 `lib/file.c` 中的其他形如 `devfile_*()` 的函数，也都提供了相应文件系统客户端的操作。这些函数的工作方式是类似的：首先在一个 `request` 变量中设定好参数，然后调用 `fsipc()` 函数将 IPC request 发送给文件系统服务器端，接着等待服务器端的响应，最后将服务器端 IPC 返回的结果进行解包和返回。回忆一下 exercise 2 后面我贴出来的那张“自底向上”示意图图，`fsipc()` 这个函数就是在 IPC 层的 `workhorse` 啦。

④`ipc_send` 和 `ipc_rcv` 就是传递文件系统调用的参数的。

⑤现在我们来到了文件系统进程中，来到了 `serve()` 函数，这个函数是一个无限循环，它不断地接收来自其他进程的 IPC 消息，根据消息中的参数，分发给相应的功能函数（这里是 `serve_read()` 函数），接着等待功能函数的返回，最后调用 IPC 将结果发送给用户进程。

⑥进入 `serve_read()` 函数，这类函数是真正的“IPC 和 file system”的交界面，它根据 IPC 请求中的参数，真正调用 file system 中的函数，完成相应的功能和返回结果。

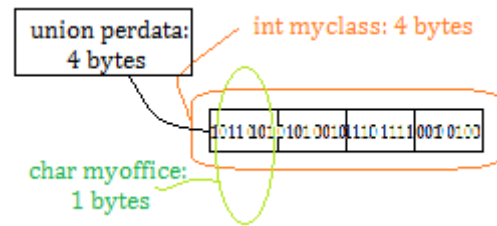
好的，经过上面的“发送请求”和“返回结果”的完整过程，我们的一个基于“远程过程调用 RPC”的文件系统功能调用就完成了。

进一步来看 IPC 的实现细节。我们回忆一下 JOS 的 IPC 机制，我们的进程之间能够同时传递两个消息：一个 32 位整数和一个页面映射。因此，在“发送文件系统功能请求”的过程中，我们将使用 32 位的整数来传递“请求类型”，“请求类型”就是相应文件系统功能调用的编号，类似我们系统调用编号、中断向量号；使用一个线性地址为 `fsreq=0x0fff f000` 的共享页面，来传递具体的调用参数，这个参数是一个“Fsipc 联合体”类型，就保存在共享页面中。在“返回结果”的过程中，我们使用 32 位整数来传递“返回值”，而对于 `FSREQ_READ` 和 `FSREQ_STAT` 还需要返回更多的数据，因此直接写在发送方发来的共享页面中。

关于什么是“联合体 union”，在网上很容易搜索到。它是 C 中的一种机制，有点类似 C++ 中的“多态”。简单介绍一下，“联合体”是一种内存使用覆盖技术，我们有时候需要使几个不同类型的变量存放到同一段内存单元中，让几个变量互相覆盖（几个不同的变量共同

占用一段内存的结构)，然后用某种类型的指针指向这一段内存，就可以按照相应的理解方式将这段内存中保存的数据“解码”出来。例如：

```
union perdata {  
    int myclass;  
    char myoffice;  
}a;
```



联合体 `perdata` 声明了一个变量 `a`，`a` 的大小等于 `perdata` 成员中最长成员的长度（在此为 `int myclass`，所以长度为 4 字节）。如果我用一个整型指针 `int *a_int = &a`；那么这个联合体就理解为 `int myclass`；如果我用一个字符型指针 `char *a_char = &a`；那么这个联合体就理解为 `char myoffice`。要注意因为成员之间互相覆盖，所以一般不可能共存。这样做据说是早期为了节省内存。

好啦，该干正事了。这个 `exercise 5` 要实现“读”操作的服务器端和客户端相应的函数，分别是 `fs/serv.c` 中的 `serve_read()` 和 `lib/file.c` 中的 `devfile_read()`。

`serve_read()` 这个函数是要将 IPC 传来的共享页面上的联合体参数 `union Fsipc *ipc`，解码出来，然后调用 `fs/fs.c` 中的 `openfile_lookup()` 函数将文件打开；调用 `file_read()` 函数，完成文件读取操作。关于服务器端的函数，都是按照一个套路来的实现的，因此不妨参考已经实现好的 `serve_set_size()` 函数。代码如下：

```
int  
serve_read(envid_t envid, union Fsipc *ipc)  
{  
    struct Fsreq_read *req = &ipc->read;  
    struct Fsret_read *ret = &ipc->readRet;  
  
    if (debug)  
        cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid,  
req->req_n);  
  
    // LAB 5: Your code here  
    struct OpenFile *o;  
    int r;  
    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)  
        return r;  
  
    int req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n; //因为最多也只能  
    读取一页，结果只能保存在共享页面中  
    if ((r = file_read(o->o_file, ret->ret_buf, req_n,  
o->o_fd->fd_offset)) < 0)  
        return r;  
  
    o->o_fd->fd_offset += r;
```

```

    return r;
}

```

`devfile_read()`这个函数是相应的客户端函数,它要考虑的是,如何将自己的参数通过 IPC 传给 `serve_read()` (通过 `fsipc()`), 因此就是将共享页面上的联合体设置一下啦, 最后它还要将 IPC 返回的读取内容复制到 `buf` 中, 代码如下:

```

static ssize_t
devfile_read(struct Fd *fd, void *buf, size_t n)
{
    int r;

    fsipcbuf.read.req_fileid = fd->fd_file.id;
    fsipcbuf.read.req_n = n;
    if ((r = fsipc(FSREQ_READ, NULL)) < 0)
        return r;

    memmove(buf, fsipcbuf.readRet.ret_buf, r);
    return r;
}

```

Exercise 6. Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.

Use `make grade` to test your code. Your code should pass "file_write" and "file_read after file_write".

接下来实现 `fs/serv.c` 中的 `serve_write()`和 `lib/file.c` 中的 `devfile_write` 函数, 思路和 exercise 5 大同小异, 代码如下:

`serve_write()`将发送来共享页面上的联合体参数 `struct Fsreq_write *req` 进行解码, 调用 `openfile_lookup()`将文件打开, 调用 `file_write` 将内容写到文件。代码如下:

```

int
serve_write(envid_t envid, struct Fsreq_write *req)
{
    if (debug)
        cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid,
            req->req_n);

    // LAB 5: Your code here.
    struct OpenFile *o;
    int r;
    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;

    int req_n = req->req_n > PGSIZE ? PGSIZE : req->req_n;
    if ((r = file_write(o->o_file, req->req_buf, req_n,
        o->o_fd->fd_offset)) < 0)

```

```

        return r;

    o->o_fd->fd_offset += r;
    return r;
}

```

`devfile_write()`函数需要注意的是，要将写入的内容先从 `buf` 中复制到共享页面的 `req_buf` 中，然后再调用 `fsipc()`。代码如下：

```

static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
    if (n > sizeof(fsipcbuf.write.req_buf))
        n = sizeof(fsipcbuf.write.req_buf);

    int r;
    fsipcbuf.write.req_fileid = fd->fd_file.id;
    fsipcbuf.write.req_n = n;

    memmove(fsipcbuf.write.req_buf, buf, n);

    r = fsipc(FSREQ_WRITE, NULL);
    return r;
}

```

2.6 Client-Side File Operations

`lib/file.c` 中的函数专门用来处理磁盘上的文件，而 Unix 的“文件描述符”并不仅限于这个功能，它还能够用来处理管道、控制台 IO 等。在 JOS 中，每种设备类型都有一个专门的 `Dev` 结构体的变量，这个结构体变量的成员是函数指针，指向该设备特定的功能函数。

文件描述符的相关定义参见文件 `inc/fd.h` 和 `lib/fd.c`。

`lib/fd.c` 中为每一个用户进程维护了一个“文件描述符表”，里面共有 `MAXFD` (=32) 个文件描述符。文件描述符表是在线性地址空间中的 `FSTABLE = 0xD000 0000` 开始的一片地址空间，在这片区域中为每个文件描述符预留一页（4KB）。当且仅当一个文件描述符被使用，相应的空间才映射到物理内存页框。

对于几乎所有的文件相关操作，用户代码都需要通过 `lib/fd.c` 中的函数来调用。另外还有 `lib/file.c` 中唯一一个用户可直接调用的函数，就是 `open()`，它的功能是创建一个文件描述符并打开指定文件。你肯定想问**为什么**是这样？看到 `lib/file.c` 开头几行：

```

struct Dev devfile =
{
    .dev_id = 'f',
    .dev_name = "file",
    .dev_read = devfile_read,
    .dev_write = devfile_write,
    .dev_close = devfile_flush,
    .dev_stat = devfile_stat,
}

```

```
.dev_trunc = devfile_trunc
};
```

这里定义了一个 Dev 结构体的变量 `devfile`，顾名思义就是和磁盘文件相关的设备了。它对 Dev 结构体中的每个函数指针赋值，赋值为 `lib/file.c` 中相应的形如 `devfile_*` 函数。这样，Dev 结构体的变量 `devfile` 就具有了操作文件的功能。因此用户不是直接调用 `lib/file.c` 中的函数，而是通过 `devfile`；当然也有例外，`open` 就是一个，至于为什么是这样我还没有想清楚。

Exercise 7. Implement `open`. The `open` function must find an unused file descriptor using the `fd_alloc()` function we have provided, make an IPC request to the file system environment to open the file, and return the number of the allocated file descriptor. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file system environment fail.

Use `make grade` to test your code. Your code should pass "open", "large file", "motd display", and "motd change".

`lib/file.c` 中的 `open()` 这个函数，作用是通过 `fd_alloc()` 找到一个未使用的文件描述符，对文件系统服务器发出一个 IPC 请求，请求打开一个文件，接着向用户返回文件描述符的序号。事实上 `open()` 和上面写的 `devfile_read()` 和 `devfile_write()` 依然是同一个套路。代码如下：

```
int
open(const char *path, int mode)
{
    struct Fd *fd_store;
    int r;

    if (strlen(path) > MAXPATHLEN)
        return -E_BAD_PATH;

    if ((r = fd_alloc (&fd_store)) < 0)
        return r;

    strcpy (fsipcbuf.open.req_path, path);
    fsipcbuf.open.req_omode = mode;

    if ((r = fsipc (FSREQ_OPEN, (void *)fd_store)) < 0) {
        fd_close (fd_store, 0);
        return r;
    }

    return fd2num (fd_store);
}
```

这里面需要给 `fsipc()` 传入一个参数 `(void *)fd_store`。为什么呢？这是希望文件系统进程将“文件描述符”的那一页映射到 `fd_store` 上。那为什么 `devfile_read()` 和 `devfile_write()`

给 `fsipc()` 传入参数为 `NULL`? 这是因为它们是建立在已经有了文件描述符 `struct Fd *fd` 的基础上, 因此不需要传回文件描述符的内存页; 而它们的数据是通过 `fsipc_buf` 来传递, `fsipc_buf` 在 `fsipc()` 中已经 `ipc_send()` 给文件系统进程, 因为是共享映射 (不是写时复制的共享哦), 因此文件系统进程对 `fsipc_buf` 的修改是对同一个物理页面的修改, 因此不需要再通过 `ipc_recv` 来重新映射。另外, 这就印证了在 MIT 材料 exercise 5 上面最后一段出现过的一句话:

...FSREQ_READ and FSREQ_STAT also return data, which they simply write to the page that the client sent its request on. **There's no need to send this page in the response IPC**, since the client shared it with the file system server in the first place. Also, in its response, FSREQ_OPEN shares with the client a new "Fd page". ...

2.7 Spawning Processes

在之前我们实现了文件系统及其 IPC 的全部功能, 现在, 我们可以试一试我们这些功能了。“进程产卵 `spawn`” 就是一个例子, 尽管我觉得这个翻译并不是很好听 T_T。这个函数允许父进程在创建子进程时, 从文件系统装载子进程可执行代码, 然后让子进程运行。`spawn()` 函数的功能, 就相当于 Unix 中先 `fork()` 然后再 `exec()`。

为什么在用户空间实现 `spawn` 要比在 kernel 中实现更容易呢? 哎没心情想了, 我觉得好像没什么区别啊。

Exercise 8. `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe`. Test your code by running the user/icode program from `kern/init.c`, which will attempt to spawn `/init` from the file system.

Use `make grade` to test your code.

`spawn()` 的代码已经提供给我们, 在 `lib/spawn.c` 中, 我觉得里面的注释几乎是 JOS 里面最长的。它的流程简单了解一下就好了 (参看注释):

- ① 打开保存可执行代码的文件;
- ② 读入 ELF 文件头;
- ③ 调用 `sys_exofork()` 创建子进程;
- ④ 设置子进程的 `Trapframe`;
- ⑤ 调用 `init_stack()` 初始化子进程的栈空间;
- ⑥ 将 ELF 文件读入子进程地址空间中的相应地址;
- ⑦ 调用 `sys_env_set_trapframe()` 设置子进程的 `eip` 和 `esp`, `eip` 是第一条指令的地址;
- ⑧ 调用 `sys_env_set_status()` 标记子进程为 `runnable` 使之可以运行。

我们要实现 `kern/syscall.c` 中 `sys_env_set_trapframe()` 这个系统调用, 很简单, 代码如下:

```
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
    // Remember to check whether the user has supplied us with a good
    // address!
```

```

struct Env *env;
int r;

if ((r = envuid2env (envuid, &env, 1)) < 0)
    return -E_BAD_ENV;

user_mem_assert (env, tf, sizeof(struct Trapframe), PTE_U);

env->env_tf = *tf;
env->env_tf.tf_cs = GD_UT | 3;
env->env_tf.tf_eflags |= FL_IF;

return 0;
}

```

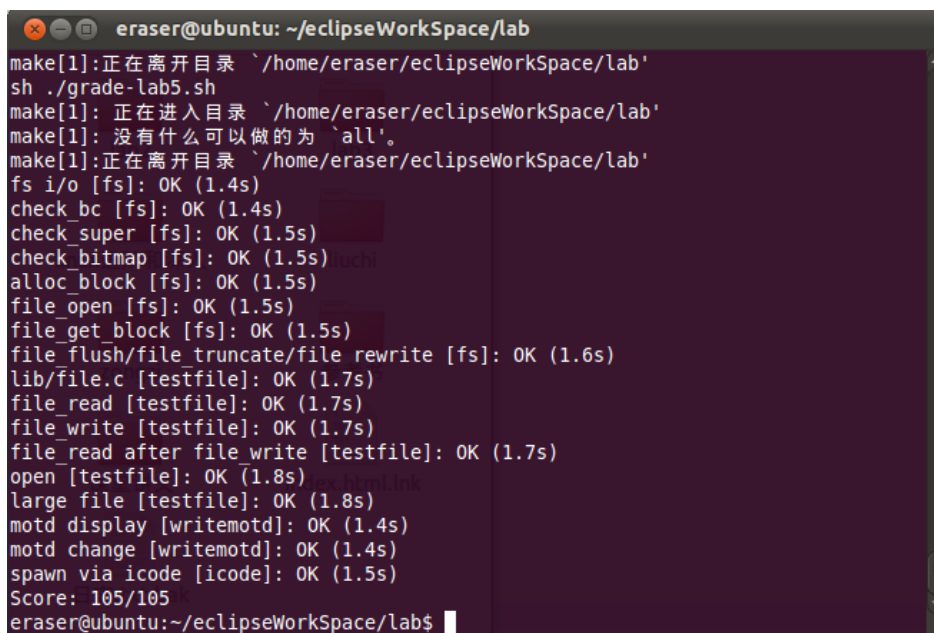
很好，提醒了我们记得检查指针是否为空，这很重要很重要很重要。另外 FL_IF 是打开外部时钟中断。别忘了添加 syscall() 中的分发语句：

```

case SYS_env_set_trapframe:
    r = sys_env_set_trapframe((envuid_t) a1, (struct Trapframe *) a2);
    break;

```

最后看看 user/icode.c，我们可以对用户进程如何调用文件系统的功能有一个了解。那么 make grade 一下，通过！



```

eraser@ubuntu: ~/eclipseWorkspace/lab
make[1]:正在离开目录 `/home/eraser/eclipseWorkspace/lab'
sh ./grade-lab5.sh
make[1]: 正在进入目录 `/home/eraser/eclipseWorkspace/lab'
make[1]: 没有什么可以做的为 `all'.
make[1]:正在离开目录 `/home/eraser/eclipseWorkspace/lab'
fs i/o [fs]: OK (1.4s)
check bc [fs]: OK (1.4s)
check_super [fs]: OK (1.5s)
check_bitmap [fs]: OK (1.5s)
alloc_block [fs]: OK (1.5s)
file_open [fs]: OK (1.5s)
file_get_block [fs]: OK (1.5s)
file_flush/file_truncate/file_rewrite [fs]: OK (1.6s)
lib/file.c [testfile]: OK (1.7s)
file_read [testfile]: OK (1.7s)
file_write [testfile]: OK (1.7s)
file_read after file_write [testfile]: OK (1.7s)
open [testfile]: OK (1.8s)
large file [testfile]: OK (1.8s)
motd display [writemotd]: OK (1.4s)
motd change [writemotd]: OK (1.4s)
spawn via icode [icode]: OK (1.5s)
Score: 105/105
eraser@ubuntu:~/eclipseWorkspace/lab$

```

内容三：遇到的困难以及解决方法

困难一：对文件系统的整个流程理解得不够透彻

这次 Lab 5 不像上次 Lab 4 一样，需要进行大量调试。很轻松，不过副作用就是对整个流程怎么走下来不是很熟悉。刚开始做的时候不太适应“自底向上”的思路，到后来才慢慢清晰，知道自己处在操作系统/文件系统哪个位置。

困难二：需要阅读大量代码

Lab 5 需要写的代码不多，但是需要阅读的代码很多。不过实践表明，这些代码不需要精读，只需要知道有这个函数、它大概干了什么就行了。

阅读代码要耐心，何况 JOS 中还有那么多注释。

内容四：收获和感想

感觉做完这个 Lab 5 收获不如前面的 Lab 多。

JOS 中的文件系统实现得比较 naïve，简化了真实操作系统的文件系统很多功能。例如，在文件缓存方面，JOS 的文件缓存空间是不加限制的，而不是采用策略决定当空间不足时保留哪些页面，写回哪些页面。而这些东西我们在原理课上曾进行详细的学习。

翻翻去年的操作系统原理课讲义，很多 trick 都没有涉及。不过既然是操统实习，简单了解原理即可，知道数组从哪里来，怎么走。至于那些 trick，其实都在 challenge 里面，但没做。

最后一个 Lab，做完之后并没有想象中“狂喜”的感觉，稍微有点累。其实还好。接下来要写其他大作业和复习期末考试了。

感谢张驰学长，感谢他的实习报告。

感谢刘驰同学，他提醒我“你怎么还没有交 Lab 5 啊”，让我亚历山大。

感谢我自己，坚持了一个学期，历经千辛万苦终于做完了！

内容五：对课程的意见和建议

（无）

内容六：参考文献

- [1] MIT 6.828 课程资料，MIT，2011；
- [2] 邵志远，JOS 实验讲义第七章，华中科技大学，2010；
- [4] 张驰，操作系统 JOS 第五次实习报告，北京大学，2011.