Lab5 设计说明

5130309029 沈匡济

Disk Access

Exercise1

在 create 一个 env 的时候判断如果是一个文件系统则给予其更高的 I/O 权限。

The Block Cache

Exercise2

这里要实现一个 buffer cache 的机制，按照文章的提示，模仿之前对 COW 机制的实现，一开始将页面设置为不可读写，在试图访问时会出发一个 pgfault，在 bc_pgfault 函数中我们对其进行处理。具体做法是申请并映射一个新的页在虚拟空间中，然后将磁盘上的内容读到该地址处，这样做的好处是节省内存，提高效率。而至于 block_flush 函数，则是对应的调用 ide_write 将数据写回到磁盘，并利用 page_map 函数将 dirty flag 去掉。

The Block Bitmap

Exercise3

这里很方便，free_block 函数中给出了 bitmap 取位的技巧，block_is_free 函数则对对应位是否为空给出了判断，只需枚举 block number 找到一个空的填上就行了。

File operations

Exercise4

根据文件地址及偏移量即可找到对应的块，如果 fileno 不在 direct[]范围内则需要视情况决定是否要分配新的 indirect block 并设置对应条目，如果设置成功则返回 indirect 数组中的第 fileno-NDIRECT 项。

file_get_block 函数是对前一个函数的封装，作用是找到一个文件中对应的块，并且如果它为空就手动分配。

Client/Server File System Access

Exercise5、6

这里要我们用 RPC 实现通信，在这个 lab 里传输层其实就是 IPC。传输的细节在 lib/fd.c 以及 inc/fs.h 中，union fsipc 中包含了一次 send-receive 中传输的信息，一开始觉得 union 结构有些奇怪，后来自己意识到这个结构本身包含了数种对 file system 的操作如 read、open、write 等，应该是因为一次只用传递一种操作的信息就可以了。

server 的代码位于 fs/serve.c 中，serve 函数中有这么一段：

```
if (req == FSREQ_OPEN) {
        r = serve_open(whom, (struct Fsreq_open*)fsreq, &pg, &perm);
} else if (req < NHANDLERS && handlers[req]) {
        r = handlers[req](whom, fsreq);
} else {
        cprintf("Invalid request code %d from %08x\n", whom, req);
        r = -E_INVAL;
}
```

```
typedef int (*fshandler)(envid_t envid, union Fsipc *req);

fshandler handlers[] = {
        // Open is handled specially because it passes pages
        /* [FSREQ_OPEN] =       (fshandler)serve_open, */
        [FSREQ_SET_SIZE] =      (fshandler)serve_set_size,
        [FSREQ_READ] =          serve_read,
        [FSREQ_WRITE] =         (fshandler)serve_write,
        [FSREQ_STAT] =          serve_stat,
        [FSREQ_FLUSH] =         (fshandler)serve_flush,
        [FSREQ_REMOVE] =        (fshandler)serve_remove,
        [FSREQ_SYNC] =          serve_sync
};
```

可以看出，程序巧妙的设置好了 handler 函数，将 ipc 收到的信息完整的传递到了 handler 中，这一部分已经帮我们实现好了，意味着传输的过程完全不用我们考虑。考虑 serve_read 函数的功能，只需找到对应的文件并调用 file_read 读出信息，再调整文件指针的偏移量即可。这里使用 OpenFile 结构体可以很方便的保存打开文件的信息。再然后是 devread 函数，即是客户端调用的读操作函数，填充 fsipcbuf 并且发送请求即可。

serve_write 与 read 原理大同小异，详见代码，不多作赘述。

这里一开始自己怎么调试也通过不了，最后发现在 pmap 中初始化映射页空间时：

```
size_t pgs = ROUNDUP(npages*sizeof(struct Page),PGSIZE);
cprintf("pages' size:0x%x Max va:0x%x\n",pgs,UVPT-UPAGES);
boot_map_region(kern_pgdir,UPAGES,pgs,PADDR(pages),PTE_P | PTE_W);
```

注释中阐明了这里是一段 user 可读的地址，但是自己并没有打上 PTE_U 的标签，导致 pgfault 的频率提高，并且发生时传入的地址显然越界，根据 bc_pgfault()中的代码，内核也因此 panic 掉了。但为什么直到这个 exercise 才发生问题呢，个人觉得是因为直到这里模拟了 RPC 才模拟出用户态操作 fs 的情景。至于为什么之前的 lab 都没出现这个问题，猜测跟 file system 独特的映射机制有关，DISKMAP 到 DISKMAP+DISKSIZE 是一段连续的 3G 的映射区间，而上述代码映射在 UPAGES 处显然是高于这段内存的，bc_pgfault 函数判断 fault 发生处不在磁盘映射范围内就会 panic，所以 UPAGES 处不该在此处发生 pgfault，要给予 PTE_U 的权限。另外，严格意义上讲，用户不应当能随意修改这块内存，所以修改方式是将 PTE_W 改为 PTE_U。

Client-Side File Operations

Exercise7

这里要编写一个客户端的 open 请求，按照注释中提到的，分为使用 fd_alloc 寻找未使用的页和发送 open 请求两部分，后者与 read、write 类似。

Spawning Processes

Exercise8

按照注释中的做法在 syscall 里添加 set_trapframe 的路由及 handler 函数。至于这里为什么要这么做，自己稍微研究了一番。lib/spawn.c 中的 spawn 里写了如下的注释：打开文件，读取 ELF 头，使用 exofork 创建子进程，为子进程初始化堆栈空间，check 过后加载 ELF 至子进程的地址空间中。比起 unix 的 exec 方式，最大的区别在于是由父进程为子进程加载 ELF 文件。

Challenge

由于期末较忙，lab5、6 的时间刚好与编译大作业、图形学大作业和期末考试重叠，且这个 lab 的 challenge 的内容明显与 exercise 不同，之前一直在完善接口，此时我并没有自信能在短时间内很好的实现任何一个 challenge 的任务，因此我选择放弃了 challenge。

Question

1.由于在 mem_init 函数中映射页时未设置用户态访问权限，因此在 exercise5 卡了一阵子，总共花了三天完成这个 lab，第一天基本都在阅读代码及复习文件系统相关知识。

2.由于上学期 cse 的基础加上课程内容对 fs 的详细讲解，我感觉自己还是有了一个较好的理解，建议方面基本没有，稍微提一个吧，我感觉具体实现的内容还可以再深入一些，比如 spawn 工作的过程，让我们参与进去一些的话 challenge 里的 exec 也会比较好写。