JOS-Lab-5 实验报告

熊伟伦
5120379076
azardf4yy@gmail.com

2014年12月10日-12月12日

Contents

1 前言

该报告描述了我在lab5实验的过程中遇到的问题与解决的方法，介绍了lab5的整体结构。指导中问题的解答参考上传的压缩包中的answers-lab5.txt文件

2 Exercise

一开始从lab4 merge后发现运行的时候少了很多输出，然后查看是被注释掉了，所以lab4的grade脚本无法通过，将这些注释变成代码后通过lab4,。猜测主要原因是这个lab与进程切换关系不大，所以没有必要输出进程。

课程网站资料介绍这个文件系统单独使用一个env管理文件系统，与CSE课程有所不一样，但继续看资料发现整个文件系统的结构和CSE的比较相似(毕竟都是模仿UNIX的)，因此这些文件系统的基础知识很多都有所了解。

但后来又发现需要实现的部分和CSE的lab不一样，CSE是实现文件系统本身，而OS是实现文件系统的调用的几个接口，系统本身已经帮我们实现好了。

2.1 Exercise-1

Exercise-1需要在系统init的时候创建创建一个env并且提供更高的I/O权限，注释介绍的很清楚直接判断if(type == ENV_TYPE_FS)。至于eflags怎么修改，我查看了inc/mmu.h，里面有个注释

inc/mmu.h

```
#define FL_IOPL_MASK 0x00003000
// I/O Privilige Level bitmask
```

因此这个exercise实现就很简单了。

kern/env.c

```
if (type == ENV_TYPE_FS)
    e->env_tf.tf_eflags |= FL_IOPL_MASK;
```

这里有个问题就是lab4里面在init里面创建的8个idle env到底还要不要，不过因为有调度，我暂时先留着了。

2.2 Exercise-2

这个Exercise开始绕了点弯(主要是API参数名看错了),深感助教的注释写的太详细了。

这两个函数，一个是从硬盘读一个sector的数据到内存，另一个数flush写回硬盘，当然实际操作都是操作整个sector所在的block。

首先是bc_pgfault函数：

kern/bc.c

```
addr = ROUNDDOWN(addr, PGSIZE);
r = sys_page_alloc(0, addr, PTE_W | PTE_U | PTE_P);
if (r < 0)
    panic("pc_pgfault: can't alloc page\n");
r = ide_read(blockno*BLKSECTS, addr, BLKSECTS);
if (r < 0)
    panic("bc_pagfault: ide_read error\n");
```

首先将addr对齐PGSIZE，分配一个page作为cache，然后读取硬盘写入到cache中。这里坑到我十来分钟的是ide_read的最后一个参数是说操作多少个sector也就是多少个512，需要写8个512。我以为是直接写BLKSIZE也就是4096。好在ide_read里面会判断第三个参数是不是少于256，否则我可能要调很久。

kern/bc.c

```
addr = ROUNDDOWN(addr, PGSIZE);
if (va_is_mapped(addr) && va_is_dirty(addr)) {
    if (ide_write(blockno*BLKSECTS, addr, BLKSECTS) < 0)
        panic("flush_block: ide_write error");
    if (sys_page_map(0, addr, 0, addr, PTE_SYSCALL) < 0)
        panic("flush_block: sys_page_map error");
}
```

第二个函数flush_block，首先对齐PGSIZE，判断这个sector所在的block是不是被映射并且是不是被修改了，然后写回到硬盘，再通过sys_page_map把dirty flag去掉，这里用sys_page_map去除flag总感觉很奇怪，但根据课程网站资料我就这样写了，自己再map自己一遍，目的只是为了修改flag，为什么不直接修改呢。

另外我看了下PTE_SYSCALL，它除了P,W,U外还包含了一个PTE_AVAIL，看注释是说user env进行硬件IO操作的位。

还有个想吐槽的地方是为何叫bc_pgfault，明明该叫bc_read或者bc_load。

2.3 Exercise-3

这里需要模仿free_block实现一个alloc_block，比较简单。block_is_free都帮我写好了，找到之后逆一下bitmap的位然后flush_block，再返回找到的blockno即可。

fs/fs.c

```
int
```

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

2.4 Exercise-4

感觉没必要把这两个函数分开，不过注释写的很详细，基本按照注释来。

首先是file_block_walk，根据给定的blockno返回一个文件对应的blockno的位置，如果在indirect的位置并且没有初始化，初始化位置了的话就alloc一个给indirect然后返回。

大致流程如下代码，很清晰。

fs/fs.c

```
static int
file_block_walk(struct File *f, uint32_t filebno,
uint32_t **ppdiskbno, bool alloc)
{
    // OUT RANGE
    if(filebno >= NDIRECT + NINDIRECT)
        return -E_INVAL;

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

    // INDIRECT
    // Need alloc
    if(f->f_indirect == 0)
    {
        if(alloc == 0)
            return -E_NOT_FOUND;
        int r = alloc_block();
        if(r < 0)
            return -E_NO_DISK;
        memset(diskaddr(r), 0, BLKSIZE);
        f->f_indirect = r;
        flush_block(diskaddr(r));
    }
```

```
29      uint32_t* indirect = diskaddr(f->f_indirect);
30      *ppdiskbno = &(indirect[filebno-NDIRECT]);
31      return 0;
32  }
```

file_get_block这个函数把上面的函数包装下，把位置传给char **blk，没有什么需要说明的。

fs/fs.c

```
1   int
2   file_get_block(struct File *f, uint32_t filebno, char **blk)
3   {
4       uint32_t * ppdiskbno = NULL;
5       // Out of range
6       if(filebno >= NDIRECT + NINDIRECT)
7           return -E_INVAL;
8
9       int r = file_block_walk(f, filebno, &ppdiskbno, 1);
10      if(r < 0)
11          return r;
12
13      // need alloc block point's block
14      if(*ppdiskbno == 0)
15      {
16          r = alloc_block();
17          if(r < 0)
18              return -E_NO_DISK;
19          *ppdiskbno = r;
20          memset(diskaddr(r), 0, BLKSIZE);
21          flush_block(diskaddr(r));
22      }
23      *blk = diskaddr(*ppdiskbno);
24      return 0;
25  }
```

这个lab良心的就是把剩下的一系列file operation都帮我们实现了。Challenge我选择了这些operation原子性的challenge，但没有实现只说了下实现的方法再在需要写代码的地方加了点注释(详见本文档Challenge章节)。

2.5 Exercise-5和Exercise-6

这里开始需要使用IPC，从其他Env调用文件系统，这调用方法与常用的文件系统有所不一样。

serve_read和serve_write函数基本就是传一传参数，调用下其他接口，没什么好说的。需要注意的是serve_read一次最多读一个BLOCKSIZE也就是一个PGSIZE(4096 byte)，因此假如req_n大于这个数字的话依然最多只能读4096个。不贴代码了。

2.6 Exercise-7和Exercise-8

首先需要完成lib/file.c的open，这里查看了下fd的结构，和fd相关的几个接口，对着注释就能写了。

然后实现kern/syscall.c里的set_trapframe。一开始我写了半天怎么都转不到这个函数里。后来才发现原来merge之后路由里面并没有添加SYS_env_set_trapframe。坑了快半个小时。

但是后来进入了这个函数还是不能跳转到init的umain，我都设置了FL_IF和protection_level到3。然后改了改去突然又能进了，代码实际效果并没有变动。我猜测应该是没有make clean去掉之前的fs相关的内容，就跟课程资料最上面一部分说的那样。代码不贴了，详见kern/syscall.c。

3 Challenge

关于这次Challenge，看了一遍所有的Challenge感觉要实现都需要完成好多工作，比exercise麻烦多了。并且到学期末还有其他几门课的大作业。

因此我选择了对文件实现原子操作的Challenge，但并没有去实现实际功能，在一些部分写了一些注释说明了下保证文件操作的原子性需要做的一些事情，这个也不太好测，因此做了一些说明注释性的东西。

设计的思路类似于数据库的原子操作，使用log保存一段记录。

以file_create操作为例，如下代码所示。在操作开始的地方write_log保存这次需要完成的操作的信息到log文件中，并且write_log需要立即刷入硬盘。然后进行操作，在file_flush之后再调用write_log说明这次操作已完成。

fs/fs.c

```
1  int
2  file_create(const char *path, struct File **pf)
3  {
4      // for Challenge
5      write_log("[num]: create begin: xxx");
6      char name[MAXNAMELEN];
7      int r;
8      struct File *dir, *f;
9
10     if ((r = walk_path(path, &dir, &f, name)) == 0)
11         return -E_FILE_EXISTS;
12     if (r != -E_NOT_FOUND || dir == 0)
13         return r;
14     if ((r = dir_alloc_file(dir, &f)) < 0)
15         return r;
16     strcpy(f->f_name, name);
17     *pf = f;
```

```
18      file_flush(dir);
19      write_log("[num]: create finish: xxx");
20      return 0;
21  }
```

在每次系统启动的时候查看write_log查看是否有begin但是没finish的操作，如果有的话再进行操作的还原再现。关键一步是要求write_log("finish")一定要在file_flush之后，防止没有file_flush的时候crash了系统却认为已经完成了整个操作。

其他的文件操作都是用大同小异，是用log的方法实现文件操作的原子性。