操统实习报告 5

1000010382 颜悦

实验报告分为三个部分：
Exercises, Questions, Challenges

Exercises

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.

根据提示，将 eflags 的 IOPL 位置位即可，这里设为了 3，根据之后的测试，没有问题。

Kern/env.c

```
    if(e->env_type == ENV_TYPE_FS)
        e->env_tf.tf_eflags |= FL_IOPL_3;
```

Exercise 2. Implement the bc_pgfault functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one your 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.

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

函数 ide_read(uint32_t secno, void *dst, size_t nsecs)在 fs/ide.c 中定义，表示从编号为 secno 的 sector 开始，读入 nsecs 个 sector， 到 dst。
关于 block cache 和内存对应地址的对应关系以及 pgfault 的处理方式，JOS 处理的比较简单，材料中说的很清楚：

We reserve a large, fixed 3GB region of the file system environment's address space, from 0x10000000 (DISKMAP) up to 0xD0000000 (DISKMAP+DISKMAX), as a "memory mapped" version of the disk.

Of course, it would be unreasonable to read the entire disk into memory, so instead we'll implement a form of demand paging, wherein we only allocate pages in the disk map region and

read the corresponding block from the disk in response to a page fault in this region. This way, we can pretend that the entire disk is in memory.

还有几个要懂的宏：
BLKSECTS, BLKSIZE, SECSIZE
在注释中有详细说明
理解看懂之后，代码就比较简单了：

Fs/bc.c

```
    // LAB 5: you code here:
    addr = ROUNDDOWN(addr, PGSIZE);
    sys_page_alloc(0, addr, PTE_P | PTE_W |PTE_U);
    ide_read(blockno*BLKSECTS, addr, BLKSECTS);
```

Exercise 3. 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/spawnhello program from kern/init.c, which will attempt to spawn /hello from the file system.

Use make grade to test your code.

按照提示完成即可

```
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
    // LAB 5: Your code here.
    // Remember to check whether the user has supplied us with a good
    // address!
    struct Env * e;
    int res = envid2env(envid, &e, 1);
    if(res < 0) return -E_BAD_FREE;
    //if(e->env_status == ENV_FREE);
//      return -E_BAD_ENV;
//  if((curenv->env_id != e->env_parent_id) && (curenv->env_id !=
e->env_id))
//      return -E_BAD_ENV;
    memcpy(&(e->env_tf), tf, sizeof(struct Trapframe));
    e->env_tf.tf_cs |= 3;
    e->env_tf.tf_eflags |= FL_IF;
    return 0;
    //panic("sys_env_set_trapframe not implemented");
}
```

Exercise 4. Change duppage in lib/fork.c to follow the new convention. If the page table entry has

the PTE_SHARE bit set, just copy the mapping directly. (You should use PTE_SYSCALL, not 0xfff, to mask out the relevant bits from the page table entry. 0xfff picks up the accessed and dirty bits as well.)

Likewise, implement copy_shared_pages in lib/spawn.c. It should loop through all page table entries in the current process (just like fork did), copying any page mappings that have the PTE_SHARE bit set into the child process.

Fork 函数的内容加一个判断即可，注意要按照提示 perm 为 pte & PTE_SYSCALL

Lib/fork.c

```
    pte_t pte = uvpt[PGNUM(addr)];
    if(pte & PTE_SHARE){
        if((r = sys_page_map(0,addr, envid, addr, pte & PTE_SYSCALL)) <
0){
            panic ("duppage: error at lab5");
        }
    }
```

这里要用好几个宏(在 lab4 中已经使用过)，uvpt 是 page table 的虚拟地址，uvpd 是 page directory 的虚拟地址，for 循环中，分别检查了 page table entry 和 page directory entry 的对应位：

Inc/mmu.h

```
// construct linear address from indexes and offset
#define PGADDR(d, t, o) ((void*) ((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

Inc/memlayout.h

```
extern volatile pte_t uvpt[];     // VA of "virtual page table"
extern volatile pde_t uvpd[];     // VA of current page directory
```

再来看代码：

Lib/spawn.c

```
// Copy the mappings for shared pages into the child address space.
static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.
    int pn ;
    for (pn = 0; pn < PGNUM(USTACKTOP); pn++){
            if (((uvpd[PDX(PGADDR(0,pn,0))]&PTE_P) &&
(uvpd[PDX(PGADDR(0,pn,0))]&PTE_U))
                && ((uvpt[pn]&PTE_P) && (uvpt[pn]&PTE_U) &&
(uvpt[pn]&PTE_SHARE))) {
            sys_page_map(0, (void *)PGADDR(0,pn,0), child, (void
*)PGADDR(0,pn,0), uvpt[pn]&PTE_SYSCALL);
```

```
        }
    }
    return 0;
}
```

Exercise 5. In your kern/trap.c, call kbd_intr to handle trap IRQ_OFFSET+IRQ_KBD and serial_intr to handle trap IRQ_OFFSET+IRQ_SERIAL.

没什么好说的…

Kern/trap.c

```
    if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
        lapic_eoi();
        kbd_intr();
        return;
    }
```

QEUSTIONS

1. Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?

2. How long approximately did it take you to do this lab?

3. We simplified the file system this year with the goal of making more time for the final project. Do you feel like you gained a basic understanding of the file I/O in JOS? Feel free to suggest things we could improve.

Answer:

1. 我们不需要额外的操作，在 trap 切换到内核的时候，压栈时会压入 eflags(包括 IOPL 位内容)，在恢复时，执行 iret 指令时，会弹栈恢复 eflags 的内容。
2. 写了大半天，也不是一直写，有 4,5 个小时吧，不包括 challenge。
3. 我觉得虽然做完了，但也不是很了解唉，建议把 lab4 内容缩减一些，增加一些 lab5 文件系统的内容。

CHALLENGES

Challenge! Extend the file system to support write access. Here are a few points you need to consider:

1. Use the block bitmap starting at block 2 to keep track of which disk blocks are free and which are in use. Look at fs/fsformat.c to see how the bitmap is initialized.

2. Make use of the alloc argument in file_block_walk. In file_get_block, allocate new disk blocks as necessary.
3. In your block cache, use the VM hardware (the PTE_D "dirty" bit in the uvpt entry) to keep track of whether a cached disk block has been modified, and thus needs to be written back to the disk.
4. Handle O_CREAT and O_TRUNC open modes in serve_open.
5. Handle more file system IPC requests, such as FSREQ_SET_SIZE, FSREQ_WRITE, FSREQ_FLUSH, FSREQ_REMOVE and FSREQ_SYNC, in fs/serv.c. We have defined the argument for these calls for you in inc/fs.h. Also, write the corresponding service routines in fs/fs.c and hook them to client stubs in lib/file.c.
6. For more information about the file system's on-disk structure, read inc/fs.h and fs/fsformat.c. You may also refer to last year's lab 5 text.

做这个 challenge 参考一些往年的代码，有一些需要的内容是直接 copy 的，有一些是自己写的，前四步自己基本实现了，5,6 两步参考的较多，主要以看懂代码了解为主。
要测试，需要复制往年的 test.c 的测试程序，以及修改 makefile 和 makegrade5 文件，比较麻烦由于有源代码，很多是学习了解的，这里就不做测试了。

首先是 bitmap 机制的建立。
Bitmap 块对应于第二个 block 的地址。
这里复制了往年代码中的 check_bitmap 和 block_is_free 函数。
在 fs_init 中加入了 check_bitmap 检测函数。
运行结果：

```
Device 1 presence: 1
superblock is good
bitmap is good
```

我们建立起了 bitmap 机制。

第二步，参考往年的材料，需要实现 ensure_block 和 alloc_block 两个函数，还要修改 file_block_walk。
具体的操作：
ensure_block：确保有效的映射存在，如果不存在，allocate 一个。
alloc_block: allocate block 的具体方法，遍历 bitmap 管理块，寻找一个空间块，并分配。
这里有一个 alloc_block 的测试，测试的话，需要去年的 test.c 文件以及修改 makefile 文件.

第三步，要利用 bc 的 dirty 位来实现 flush，需要实现的是：
Va_is_mapped, va_is_dirty 判断 PTE_P 和 PTE_D 位。
Flush_block：对标记了 dirty 位的 flush 回磁盘。
检测需要函数 check_bc.

第四步，在 server_open 中增加对 req->req_omode 中 O_CREAT 位和 O_TRUNC 的判断，并调用相应的处理函数 file_create 和 file_set_size.

第五部，描述中涉及到的文件都需要加以修改，这一步就是建立起服务程序的 handler，这里需要理解 C/S 模型，了解服务器与用户进程通信，完成一系列 fisipc*程序以及对应的处理程序，这一步我只是阅读了一下代码以及相关的一些资料。

做这个 challenge 可以更进一步的了解一下文件系统中 bitmap,写文件的一种基于 bc 的实现方式，以及 C/S 文件系统通信模型，花了好多时间看代码，对比新旧代码，实现上没有太完整，但还是有很多收获的。