JOS 2016 Lab 5 实习报告

陈一茹 1400012976

December 6, 2016

Contents

1 Introduction

JOS 是一个微内核的操作系统，JOS 的代码已经提供了一套简单的只读的磁盘文件系统。它利用一个专门的进程来实现与磁盘交互的功能。这个lab就是主要去实现文件系统的功能。

2 File system preliminaries

2.1 On-Disk File System Structure

大多数的文件系统把可获得磁盘区域分成两种区域: inode regions and data regions。 inode 包含文件的关键信息，data 包含文件的数据。文件和目录文件都包含一系列的数据段，这些数据段分配在多段物理内存上，可能是分散的。

2.1.1 Sectors and Blocks

Sector是硬盘的一个扇区，block是操作系统概念层次上的块。 JOS一个块有4096B。

2.1.2 Superblocks

超级块存储整个文件系统的基本信息。比如，块的大小，磁盘的大小，找到根目录的必须元数据等。

我们文件系统必须要有一个superblock,存在磁盘的第一块上，分布在struct Super in inc/fs.h上定义。

2.1.3 File Meta-data

文件元数据，在 struct File in inc/fs.h中被描述，这里包含文件的大小，类型，名字等信息。 有直接块， 直接储存块号，间接块是可以指向一个页面， 能够保存 1024个块，这样，我们的文件最多就可以达到1034个块，能够覆盖4G的内存了。

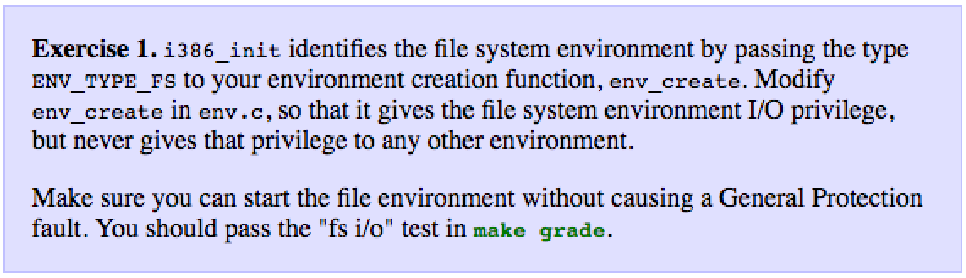
2.1.4 Directories versus Regular Files

正常文件和目录文件是用文件类型来区分的，文件系统对这两种文件的处理方式是一样的，

3 The File System

我希望文件系统能够获得访问I/O区域，我们不希望其他进程能够访问它。

3.1 Disk Access

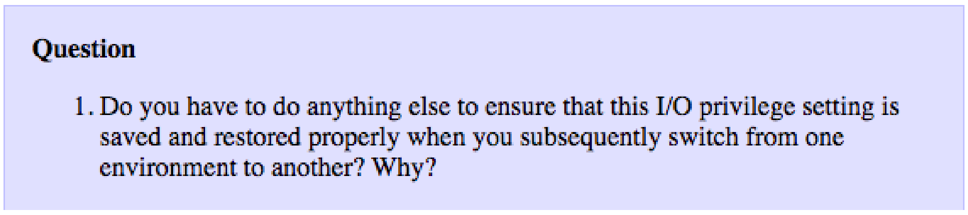


Exercise 1. `i386_init` identifies the file system environment by passing the type `ENV_TYPE_FS` to your environment creation function, `env_create`. Modify `env_create` in `env.c`, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.

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`.

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

简单修改 env_create 函数，为管理文件系统的进程(进程类型为 ENV_TYPE_FS) 设置 eflag 中 IOPL 位为 3，使其拥有访问 I/O 设备的权限。其它用户进程均为 0。



Question

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?

进程切换是在env_run()中完成的，其中 有env_pop_tf函数

```
    /
    void
env_pop_tf(struct Trapframe *tf)
{
    // Record the CPU we are running on for user-space debugging
    curenv->env_cpunum = cpunum();
    unlock_kernel();


    asm volatile(
            "\tmovl %0,%%esp\n"
            "\tpopal\n"
            "\tpopl %%es\n"
            "\tpopl %%ds\n"
            "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
```

```
		"\tiret\n"
		: : "g" (tf) : "memory");
	panic("iret failed");  /* mostly to placate the compiler */
}
```

按照我之前的分析， 在这个函数中， iret指令中恢复了eip, cs以及eflags寄存器。所以我们看到这个值能够被正确的保存和切换。

3.2 The Block Cache

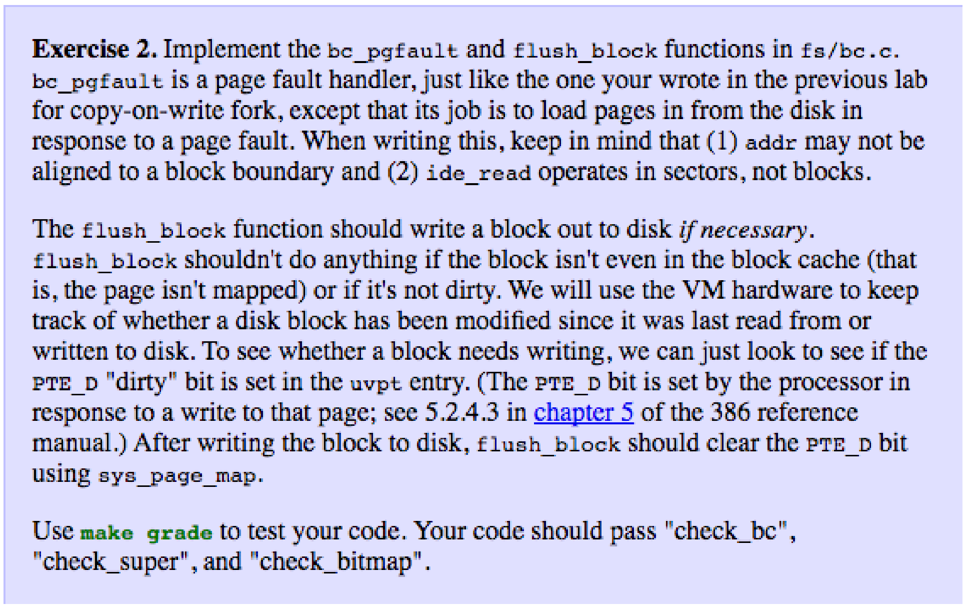
因为JOS支持的磁盘大小最大在3GB左右，所以我们可以使用类似lab4中实现fork的COW页面机制，也就是

1. 用文件系统服务进程的虚拟地址空间(4GB)对应到磁盘的地址空间上(3GB)

2. 初始文件系统服务进程里什么页面都没映射，如果要访问一个磁盘的地 址空间，则发生页错误

3. 在页错误处理程序中，在内存中申请一个块的空间映射到相应的文件系 统虚拟地址上，然后去实际的物理磁盘上读取这个区域的东西到这个内存区域上，然后恢复文件系统服务进程

这样就使用用户进程的机制完成了对于物理磁盘的读写机制，并且尽量少节 省了内存。


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 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.

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 `uvpt` 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".

实现bc_pgfault：这个函数相当于一个缺页的处理函数，当发生缺页的时候，就从磁盘上加载一个页面填上。

这中间用到fs/ide.c中的函数，用来和磁盘进行直接交互的IDE驱动

```
int
ide_read(uint32_t secno, void *dst, size_t nsecs)
int
ide_write(uint32_t secno, const void *src, size_t nsecs)
```

secno对应IDE磁盘上的扇区编号，dst为当前文件系统服务程序空间中的对 应地址，nsecs为读写的扇区数。 注意这是写的扇区，一个block是8个扇区。

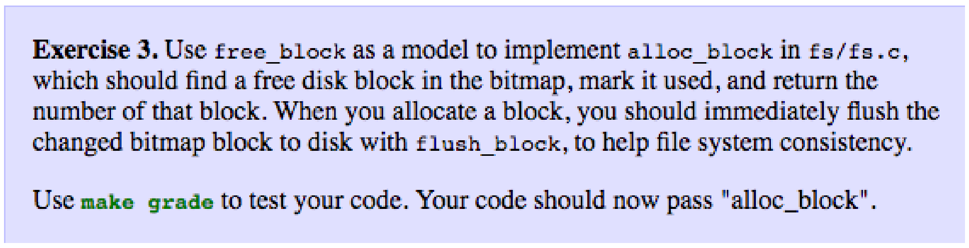
```
    // Allocate a page in the disk map region, read the contents
    // of the block from the disk into that page.
    // Hint: first round addr to page boundary. fs/ide.c has code to read
    // the disk.
    //
    // LAB 5: you code here:
    addr = (void*)ROUNDDOWN((uintptr_t)addr, PGSIZE);
    sys_page_alloc(0, addr, PTE_P | PTE_U | PTE_W);
    ide_read(blockno << 3, addr, 8);
```

flush_block函数： 如果这个页没有被map,也没有设置dirty位，那么这函数什么也不做

```
      // LAB 5: Your code here.
   addr = ROUNDDOWN(addr, PGSIZE);

   int r;

    if(va_is_dirty(addr)&&va_is_mapped(addr))
    {
        ide_write(blockno*BLKSECTS, addr, BLKSECTS);

        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
            panic("flush_block, sys_page_map: %e", r);
    }
```

3.3 The Block Bitmap

指导手册说，我们可以把bitmap 看成是一个整合好的bit的数组。



Exercise 3. Use `free_block` as a model to implement `alloc_block` in `fs/fs.c`, 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".

```
    int
alloc_block(void)
{
        // The bitmap consists of one or more blocks.  A single
             bitmap block
        // contains the in-use bits for BLKBITSIZE blocks.
            There are
        // super->s_nblocks blocks in the disk altogether.

        // LAB 5: Your code here.
        int blockno;

        for(blockno = 0; blockno < super->s_nblocks; blockno++)
        {
            if(block_is_free(blockno))
            {
                bitmap[blockno/32] ^= 1<<(blockno%32);
                flush_block(bitmap+blockno/8);
                return blockno;
            }
        }

        return -E_NO_DISK;
}
```

3.4 File Operations

- file block walk(*f, filebno, ppdiskbno, alloc)：寻找一个文件结构f中的第filebno个块指向的硬盘块编号放入ppdiskbno，即如果filebno小于NDIRECT，则返回属于f direct[NDIRECT]中的相应 链接，否则返回f indirect中查找的块。如果alloc为真且相应硬盘块不存在，则分配一个。当我们要将一个修改后的文件flush回硬盘，就需要使用这个函数找一个 文件中链接的所有磁盘块，将他们都flush

- block dir lookup(*dir, *name, **file)：这个很明显了

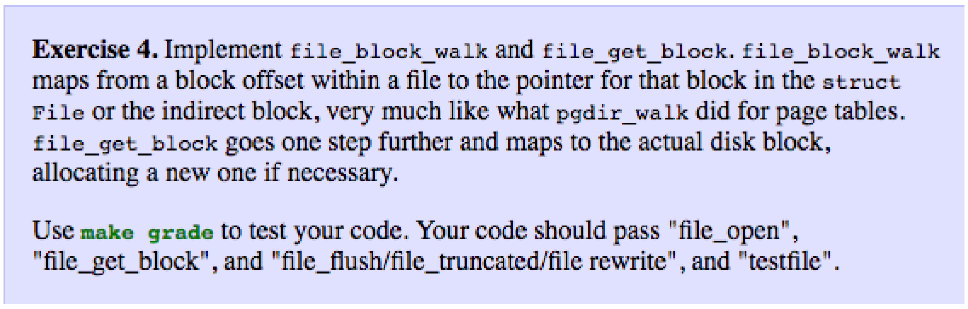
- dir alloc file(*dir, **file)：在*dir对应的File结构中分配一个File的指针链接给*file，用于添加文件的 操作。

- skip slash(*p)：用于路径中的字符串处理，调过斜杠。

- walk path(*path, **pdir, **pf, *lastlem)：*path为从根目录开始描述的文件名，如果成功找到了文件，则把相应的 文件File结构赋值给*pf，其所在目录的File结构赋值给**pdir，lastlem为 失败时最后剩下的文件名字。

- file free block(*f, filebno)：释放一个文件中的第filebno个磁盘块。此函数在file truncate blocks中被 调用

- file truncate blocks(*f, newsize)：将文件设置为缩小后的新大小，清空那些被释放的物理块。



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", and "testfile".

```
  // Find the disk block number slot for the 'filebno'th block in file 'f'.
// Set '*ppdiskbno' to point to that slot.
// The slot will be one of the f->f_direct[] entries,
// or an entry in the indirect block.
// When 'alloc' is set, this function will allocate an indirect block
// if necessary.
//
// Returns:
//      0 on success (but note that *ppdiskbno might equal 0).
//      -E_NOT_FOUND if the function needed to allocate an indirect block, but
//              alloc was 0.
//      -E_NO_DISK if there's no space on the disk for an indirect block.
//      -E_INVAL if filebno is out of range (it's >= NDIRECT + NINDIRECT).
//
// Analogy: This is like pgdir_walk for files.
// Hint: Don't forget to clear any block you allocate.
    static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
    int r;
    uint32_t *ptr;
    char *blk;

    if(filebno < NDIRECT)
    {
        ptr = &f->f_direct[filebno];
    }else if(filebno < NDIRECT + NINDIRECT)
        {
        if(f->f_indirect == 0 && alloc == 0)
            return -E_NOT_FOUND;
        else if(f->f_indirect==0)
        {
            if((r = alloc_block()) < 0)
                return -E_NO_DISK;
            f->f_indirect = r;
            memset(diskaddr(r), 0, BLKSIZE);
            flush_block(diskaddr(r));
        }
        ptr = (uint32_t*)diskaddr(f->f_indirect) + filebno -
            NDIRECT;
    }else
    {
        return -E_INVAL;
    }
    *ppdiskbno = ptr;
    return 0;
}
```

就是找到相应的槽，并用一个指针指向它，这里要注意Set '*ppdiskbno' to point to that slot.注释中的note that *ppdiskbno might equal 0应该是 **ppdiskbno ，也就是说这一块还没有的时候，这里要alloc_block()一下，我放在了下一个函数中了。

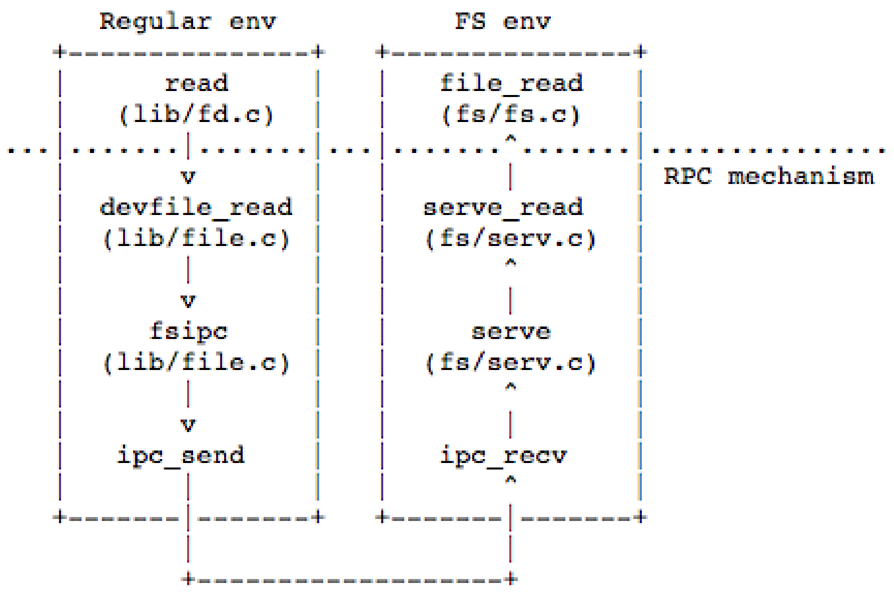
```
  // Set *blk to the address in memory where the filebno'th
// block of file 'f' would be mapped.
//
// Returns 0 on success, < 0 on error.  Errors are:
//      -E_NO_DISK if a block needed to be allocated but the
    disk is full.
//      -E_INVAL if filebno is out of range.
//
// Hint: Use file_block_walk and alloc_block.
    int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
    int r;
    uint32_t *ptr;

    if ((r = file_block_walk(f, filebno, &ptr, 1)) < 0)
        return r;
    if (*ptr == 0) {
        if((r = alloc_block())<0)
            return -E_NO_DISK;

        *ptr = r;
        memset(diskaddr(r), 0, BLKSIZE);
        flush_block(diskaddr(r));
    }
    *blk = diskaddr(*ptr);
    return 0
}
```

blk最后应该指向得到 的blk对应在文件系统地址空间中的地址，所以最后

需要用diskaddr转换。同样 和上面的函数一样要记得写回新申请的物理块数据。

3.5 The file system interface

这个部分关于RPC。



以上是RPC的一个读操作的示意。

fs/serv.c段的工作：

```
  void
serve(void)
{
        uint32_t req, whom;
        int perm, r;
        void *pg;

        while (1) {
                perm = 0;
                req = ipc_recv((int32_t *) &whom, fsreq, &perm);
                if (debug)
                        cprintf("fs req %d from %08x [page %08x: %s]\n",
                                req, whom, uvpt[PGNUM(fsreq)], fsreq);

                // All requests must contain an argument page
                if (!(perm & PTE_P)) {
                        cprintf("Invalid request from %08x: no argument page\n",
                            whom);
                        continue; // just leave it hanging...
                }

                pg = NULL;
                if (req == FSREQ_OPEN) {
                        r = serve_open(whom, (struct Fsreq_open *)fsreq, &pg, &perm);
                } else if (req < ARRAY_SIZE(handlers) && handlers[req]) {
                        r = handlers[req](whom, fsreq);
                } else {
                        cprintf("Invalid request code %d from %08x\n", req, whom);
                        r = -E_INVAL;
                }
                ipc_send(whom, r, pg, perm);
                sys_page_unmap(0, fsreq);
        }
}
```

服务器主循环会使用轮询的方式接受客户端程序的文件请求，每次

1. 从IPC接受一个请求类型req以及数据页fsreq
2. 然后根据req来执行相应的服务程序
3. 将相应服务程序的执行结果(如果产生了数据页则有pg)通过IPC发送回 调用进程
4. 将映射好的物理页fsreq取消映射

```
  inc/fs.h
 union Fsipc {
         struct Fsreq_open {
                 char req_path[MAXPATHLEN];
        int req_omode;
} open;
struct Fsreq_set_size {
        int req_fileid;
        off_t req_size;
} set_size;
struct Fsreq_read {
        int req_fileid;
        size_t req_n;
} read;
struct Fsret_read {
        char ret_buf[PGSIZE];
} readRet;
struct Fsreq_write {
        int req_fileid;
        size_t req_n;
        char req_buf[PGSIZE - (sizeof(int) + sizeof(size_t))];
} write;
```

```
struct Fsreq_stat {
        int req_fileid;
} stat;
struct Fsret_stat {
        char ret_name[MAXNAMELEN];
        off_t ret_size;
        int ret_isdir;
} statRet;
struct Fsreq_flush {
        int req_fileid;
} flush;
struct Fsreq_remove {
        char req_path[MAXPATHLEN];
} remove;
```

Fsipc的结构类型。

```
// Read at most ipc->read.req_n bytes from the current seek
   position
// in ipc->read.req_fileid.  Return the bytes read from the
   file to
// the caller in ipc->readRet, then update the seek position.
   Returns
// the number of bytes successfully read, or < 0 on error.
    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:
    // First, use openfile_lookup to find the relevant open
        file.
    // On failure, return the error code to the client with
        ipc_send.
    Struct OpenFile *o;
    int r;

    if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;

    if ((r = file_read(o->o_file, ret->ret_buf, MIN(req->req_n,
         sizeof ret->ret_buf),o->o_fd->fd_offset)) < 0)
        return r;

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

这里用同一个Union来存储发送请求和返回的信息，由于是值传递，这里不会发生覆盖错误。

```
serve_write
// Write req->req_n bytes from req->req_buf to req_fileid, starting at
// the current seek position, and update the seek position
// accordingly.  Extend the file if necessary.  Returns the number of
// bytes written, or < 0 on error.
    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 > sizeof(req->req_buf) ? sizeof(req->req_buf): 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;

}
```

这个函数和上面的一模一样。

还要实现一个函数devfile_write in lib/file.c

```
// Write at most 'n' bytes from 'buf' to 'fd' at the current seek position.
//
// Returns:
//       The number of bytes successfully written.
//       < 0 on error.
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
        // Make an FSREQ_WRITE request to the file system server.  Be
        // careful: fsipcbuf.write.req_buf is only so large, but
        // remember that write is always allowed to write *
           fewer*
        // bytes than requested.
        // LAB 5: Your code here

        if (n > sizeof (fsipcbuf.write.req_buf))
           n = sizeof (fsipcbuf.write.req_buf);

        fsipcbuf.write.req_fileid = fd->fd_file.id;
        fsipcbuf.write.req_n = n;
        memmove(fsipcbuf.write.req_buf,buf,n);
        int r;
        if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
           return r;
        return r;
}
```

这里是模仿devfile_read写的。很简单。

但是我还是在这里卡了很久，因为源代码里没有

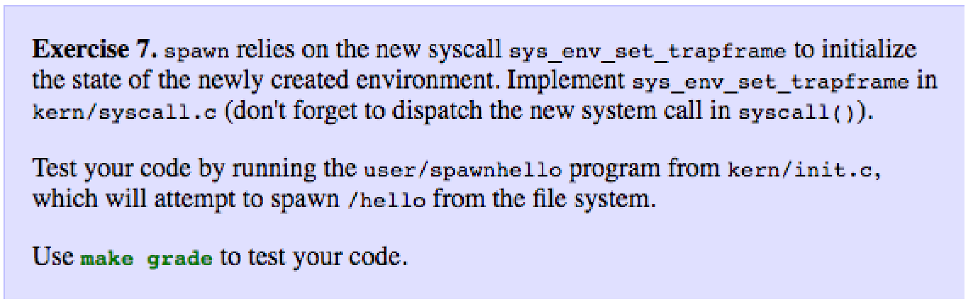
```
fshandler handlers[] = {
    // Open is handled specially because it passes pages
    [FSREQ_OPEN] =   (fshandler)serve_open,
    [FSREQ_READ] =      serve_read,
    [FSREQ_STAT] =      serve_stat,
    [FSREQ_FLUSH] =     (fshandler)serve_flush,
    [FSREQ_WRITE] =     (fshandler)serve_write,
};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

也是醉了。。。

4 Spawning Processes

Exercise 7是实现spawn，与我们熟悉的exec所不一样的地方就是spawn是由父进程为子进程load code的，而exec是子进程自己来的。所以exec更加难一点。

这其实是一个微内核的文件系统，文件系统本身作为一个进。在运行，可以支持对层次目录中的文件进行相关的文件操作read, write等。spawn从文件系统中装载并运行一个文件。其他进程通过IPC请求的方式来访问文件系统的服务。


Exercise 7. `spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement `sys_env_set_trapframe` in `kern/syscall.c` (don't forget to dispatch the new system call in `syscall()`).

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.

JOS 实现了 spawn 函数，用于创建一个新进程来运行磁盘中的可执行文件。Spawn 实现在用户态，使用 exofork 创建进程，从磁盘读取 elf 格式文件，然后 类似于 load_icode 将可执行文件加载到内存。

```
// Set envid's trap frame to 'tf'.
// tf is modified to make sure that user environments always run at code
// protection level 3 (CPL 3) with interrupts enabled.
//
// Returns 0 on success, < 0 on error.  Errors are:
//	-E_BAD_ENV if environment envid doesn't currently exist,
//		or the caller doesn't have permission to change envid.
	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;
	if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -E_BAD_ENV;
	if ((uintptr_t)tf >= UTOP || !page_lookup(curenv->env_pgdir, tf, 0)) return -E_INVAL;
	tf->tf_cs |= 3;
	tf->tf_eflags |= FL_IF;
	e->env_tf = *tf;
	return 0;
}
```

这里就是实现了一个新的系统调用。设置某个进程 的 Trapframe 为给定值。

```
	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;
    if (envid2env(envid, &e, 1) == -E_BAD_ENV) return -
        E_BAD_ENV;
    if ((uintptr_t)tf >= UTOP || !page_lookup(curenv->env_pgdir
        , tf, 0)) return -E_INVAL;
    tf->tf_cs |= 3;
    tf->tf_eflags |= FL_IF;
    e->env_tf = *tf;
    return 0;
}
```

这里还有一个问题，就是 faultio，我卡了好久没有能过，后来发现这样的问题：

The TSS contains a 16-bit pointer to I/O port permissions bitmap for the current task. This bitmap, usually set up by the operating system when a task is started, specifies individual ports to which the program should have access. The I/O bitmap is a bit array of port access permissions; if the program has permission to access a port, a "0" is stored at the corresponding bit index, and if the program does not have permission, a "1" is stored there. The feature operates as follows: when a program issues an x86 I/O port instruction such as IN or OUT (see x86 instruction listings), the hardware will do an I/O privilege level (IOPL) check to see if the program has access to all I/O ports. If the CPL of the program is numerically greater than the IOPL (the program is less-privileged than what the IOPL specifies), the program does not have I/O port access to all ports. The hardware will then check the I/O permissions bitmap in the TSS to see if that program can access the specific port in the IN or OUT instruction. If the bit in the I/O port permissions bitmap is clear, the program is allowed access to this port, and the instruction is allowed to execute. If the bit is set, the program does not have access and the processor generates a general protection fault. This feature allows operating systems to grant selective port access to user programs.

这是在wiki上找的资料，eflags中的权限值相等的时候，我们还可以选择性的给权限，就是使用这个 I/O bitmap （ a bit array of port access permissions）。

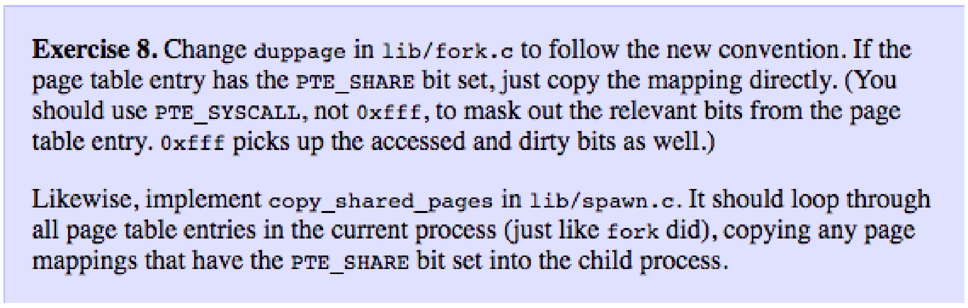
其实这个问题是在JOS中帮我们设置好的，但是我写lab4的Ex4时改函数的时候把这个删掉了，所以这里出了问题。

这里应该是这样的：

```
        thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);
```

这句话就是将IN和OUT的值设成1。使得protection fault出现了。

4.1 Sharing library state across fork and spawn


Exercise 8. 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.

lib/fork.c 如果是shared的，就把他们直接map一下。

```
        if (perm & PTE_SHARE) {
          if ((r = sys_page_map(0, (void*)(pn * PGSIZE), envid, (void*)(pn * PGSIZE), perm)))
            return r;
          return 0;
      }
```

lib/spawn.c 类似于fork，寻找所有的，将SHARED的直接map.

```
    // Copy the mappings for shared pages into the child address space.
    static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.
    uintptr_t p = 0;
    while (p < UTOP) {
        if (!uvpd[p >> 22]) {
            p += PGSIZE << 10;
            continue;
        }
        if (p != UXSTACKTOP - PGSIZE && uvpt[p >> 12]) {
            int perm = uvpt[p >> 12] & 0xfff;
            if (perm & PTE_SHARE)
                sys_page_map(0, (void*)p, child, (void*)p, perm);
        }
        p += PGSIZE;
    }
    return 0;
}
```

5 The keyboard interface

Exercise 9. 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`.

```
        // Handle keyboard and serial interrupts.
    // LAB 5: Your code here.
    if (tf->tf_trapno == IRQ_OFFSET+IRQ_KBD) {
        kbd_intr();
        return;
    }

    if (tf->tf_trapno == IRQ_OFFSET+IRQ_SERIAL) {
        serial_intr();
        return;
    }
```

在 trap_dispatch中添加两个系统调用。

6 The Shell

Exercise 10.

The shell doesn't support I/O redirection. It would be nice to run `sh <script` instead of having to type in all the commands in the script by hand, as you did above. Add I/O redirection for < to `user/sh.c`.

Test your implementation by typing `sh <script` into your shell

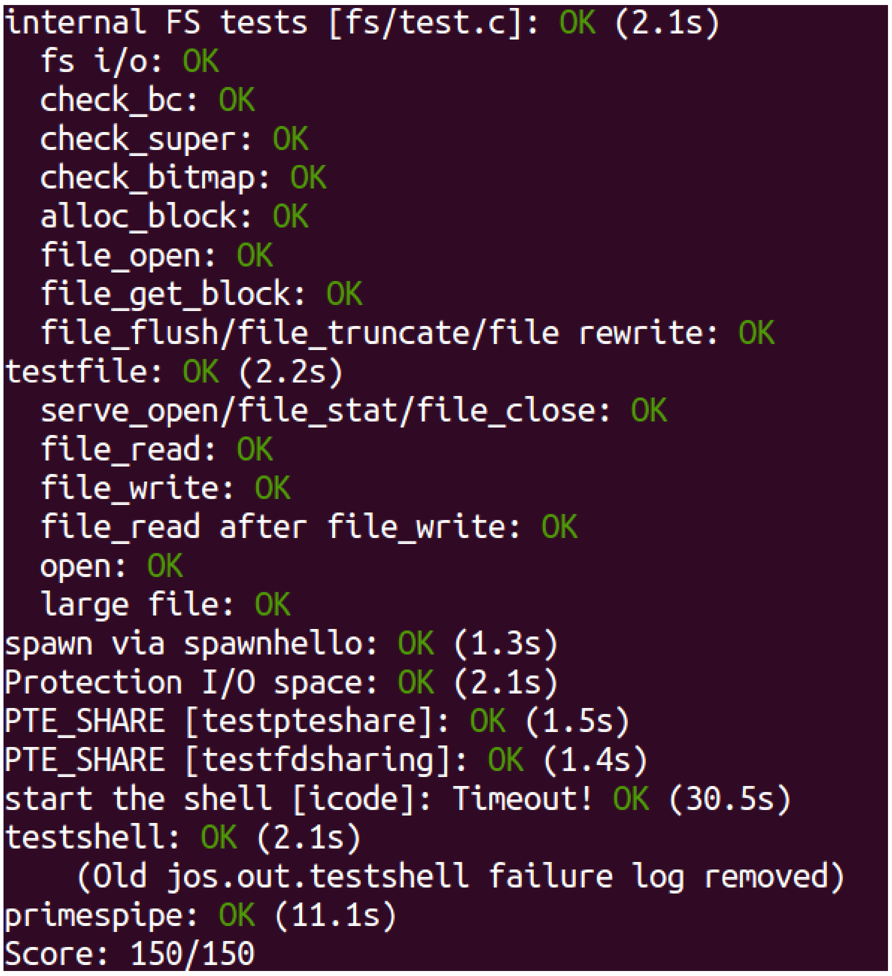
Run `make run-testshell` to test your shell. `testshell` simply feeds the above commands (also found in `fs/testshell.sh`) into the shell and then checks that the output matches `fs/testshell.key`.

这个exercise要求支持shell的操作，需要实现 sh¡script的函数。

```
    case '<':   // Input redirection
        // Grab the filename from the argument list
        if (gettoken(0, &t) != 'w') {
            cprintf("syntax error: < not followed by word\n");
            exit();
        }
        // Open 't' for reading as file descriptor 0
        // (which environments use as standard input).
        // We can't open a file onto a particular descriptor,
        // so open the file as 'fd',
        // then check whether 'fd' is 0.
        // If not, dup 'fd' onto file descriptor 0,
        // then close the original 'fd'.

        // LAB 5: Your code here.
        if ((fd = open(t, O_RDONLY)) < 0) {
            cprintf("open %s for read: %e", t, fd);
            exit();
        }
        if (fd != 0) {
            dup(fd, 0);
            close(fd);
        }
        break;
```

这个和¿重定向输出，是一样的，改一点地方就可以了。主要就是判断一下文件是否可读，然后 dup(fd, 0)重定向标准输入。很简单。

7 Complete!



```
internal FS tests [fs/test.c]: OK (2.1s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
  alloc_block: OK
  file_open: OK
  file_get_block: OK
  file_flush/file_truncate/file rewrite: OK
testfile: OK (2.2s)
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
spawn via spawnhello: OK (1.3s)
Protection I/O space: OK (2.1s)
PTE_SHARE [testpteshare]: OK (1.5s)
PTE_SHARE [testfdsharing]: OK (1.4s)
start the shell [icode]: Timeout! OK (30.5s)
testshell: OK (2.1s)
    (Old jos.out.testshell failure log removed)
primespipe: OK (11.1s)
Score: 150/150
```

This completes the lab!!!