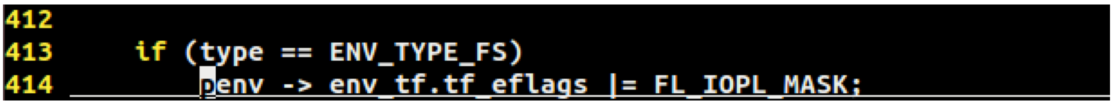
实验报告

Lab4

姓名：朱瑞媛

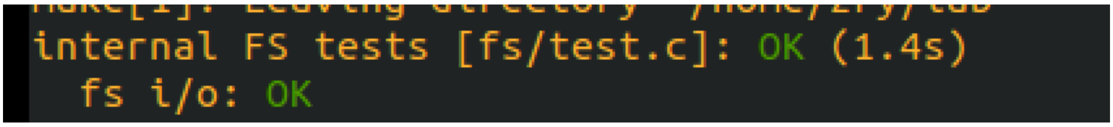
学号：**14307130373**

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

在env_create函数中修改进程的eflag值。



```
make[1]: Leaving directory `/home/zfy/lab
internal FS tests [fs/test.c]: OK (1.4s)
  fs i/o: OK
```

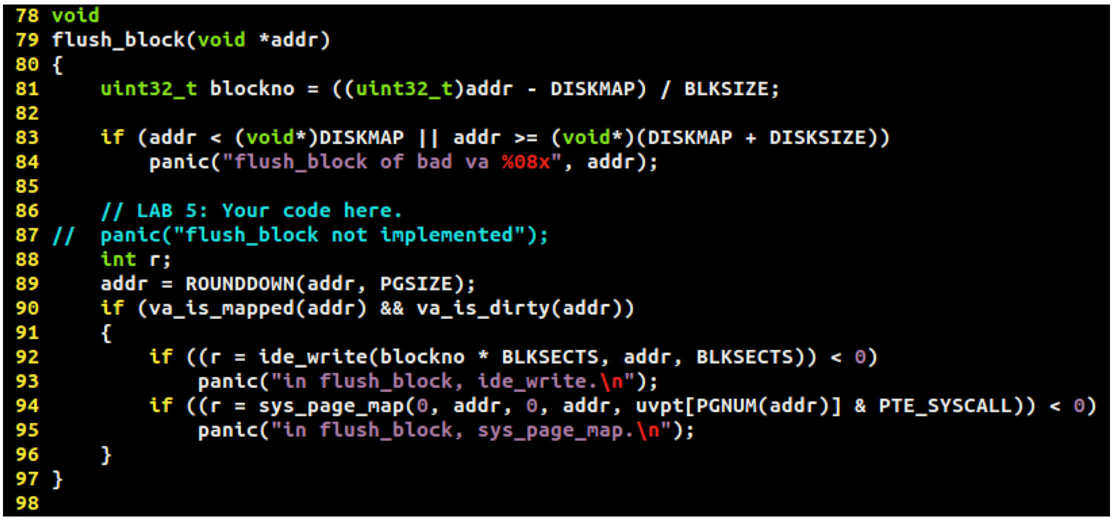
通过测试。

Question 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?

在进程切换时调用了env.pop_tf函数，其中进行了寄存器的恢复，在iret指令中恢复了eip，cs，eflags等寄存器。

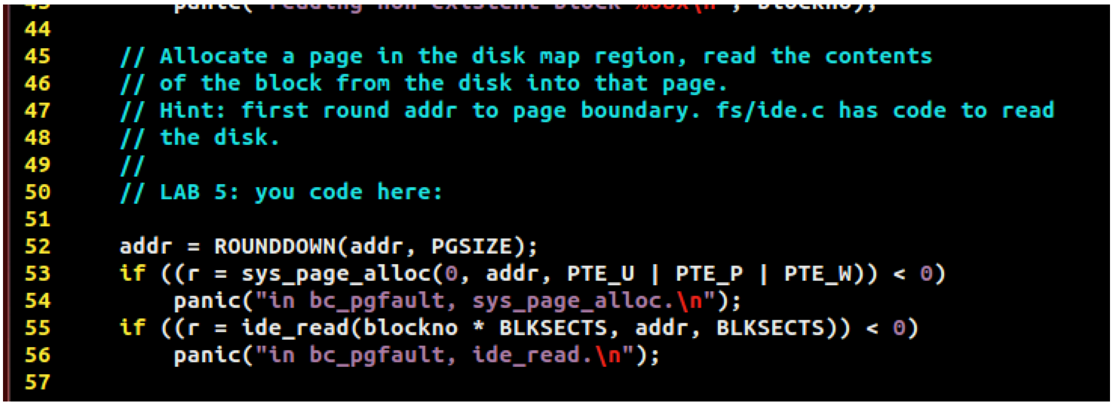
Exercise 2. Implement the bc_pgfault and flush_block functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (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".



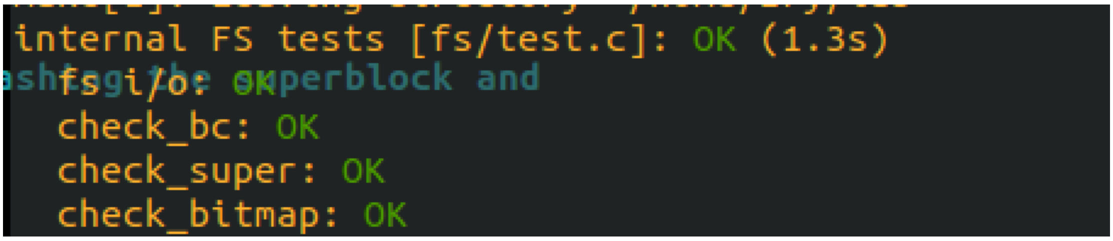
```
void
flush_block(void *addr)
{
    uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

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

    // LAB 5: Your code here.
//  panic("flush_block not implemented");
    int r;
    addr = ROUNDDOWN(addr, PGSIZE);
    if (va_is_mapped(addr) && va_is_dirty(addr))
    {
        if ((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
            panic("in flush_block, ide_write.\n");
        if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
            panic("in flush_block, sys_page_map.\n");
    }
}

```

先根据地址计算对应的blockno，然后然后检查正确性，最后判断是否是脏块，如果是则写回磁盘并清除dirty位。



```
        panic("reading non-existent block %08x\n", blockno);

    // 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 = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, addr, PTE_U | PTE_P | PTE_W)) < 0)
        panic("in bc_pgfault, sys_page_alloc.\n");
    if ((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
        panic("in bc_pgfault, ide_read.\n");

```

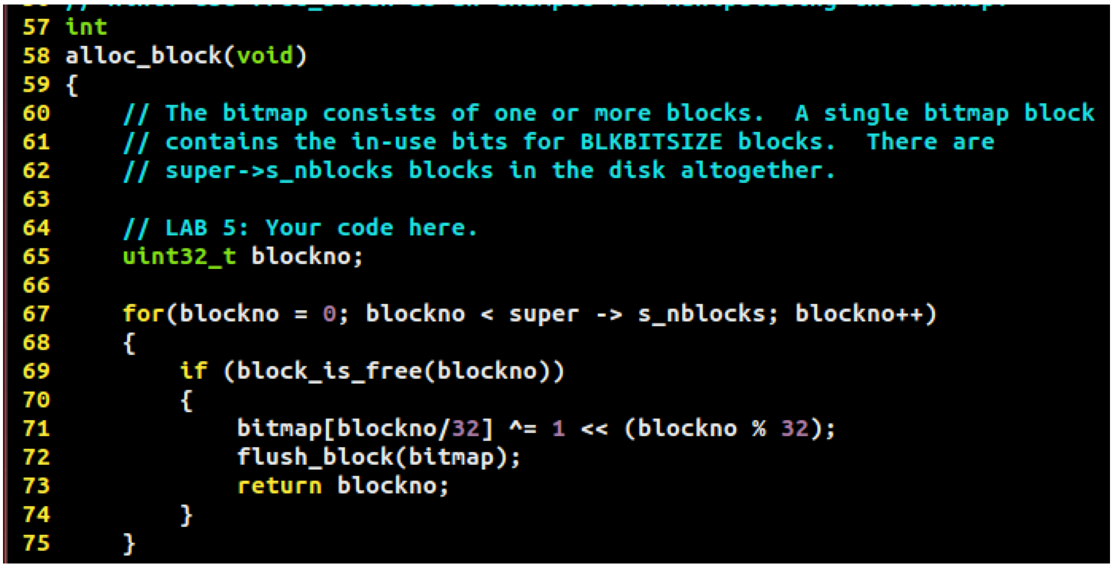
先根据地址计算出对应的blockno，然后检查正确性包括地址是否在映射范围内、对应的block是否存在等。



```
internal FS tests [fs/test.c]: OK (1.3s)
  fs i/o: OK
  check_bc: OK
  check_super: OK
  check_bitmap: OK
```

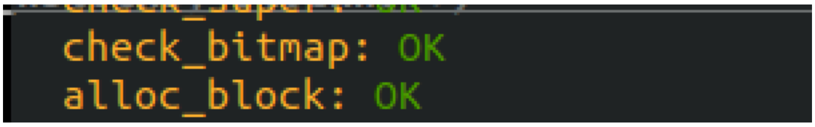
通过测试。

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".



```c
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.
	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;
		}
	}
```

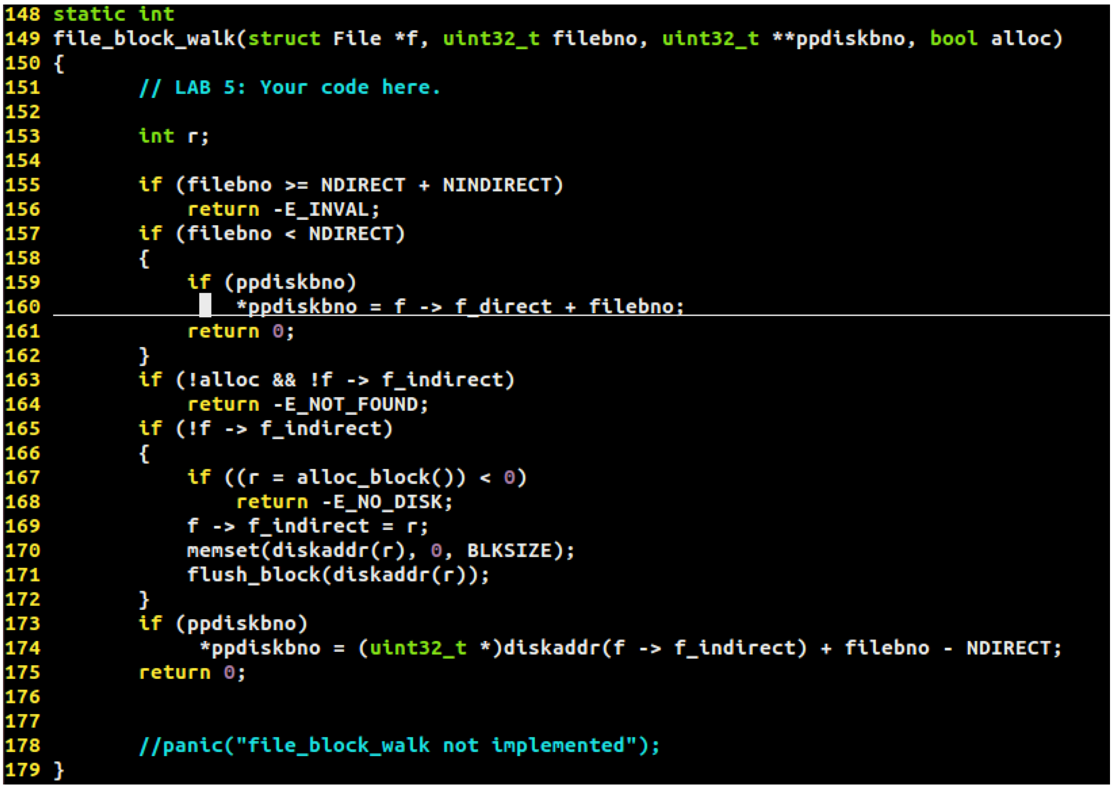
以free_block为参考实现alloc_block，功能是在位图中查找1个空闲磁盘块，标记为占用并返回块序号。当分配1个块时，为了维护文件系统的一致性，需要快速地使用flush_block函数写回你对位图的修改。



```
  check_bitmap: OK
  alloc_block: OK
```

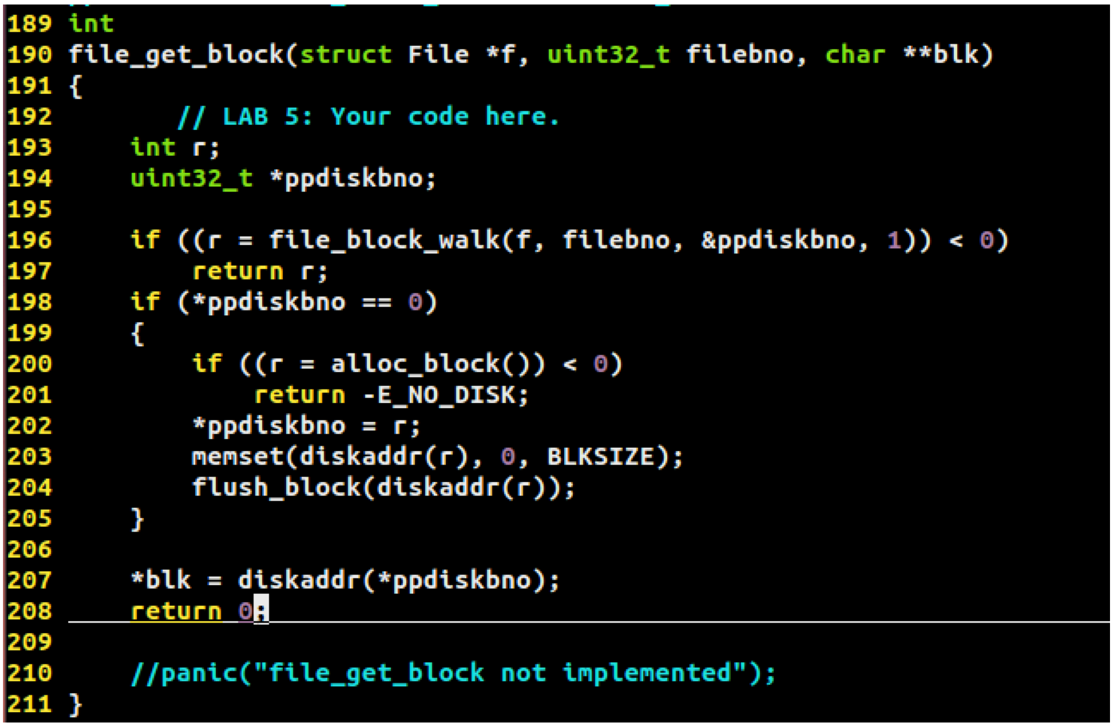
通过测试。

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.



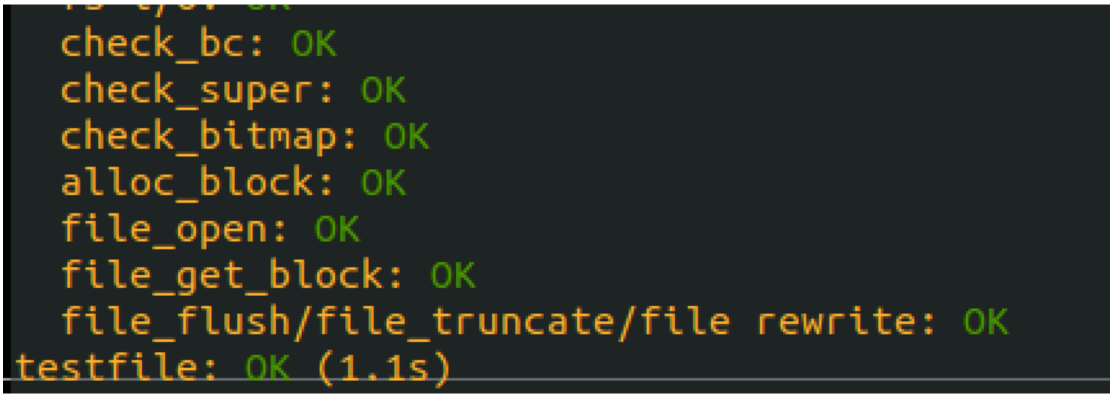
```
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
        // LAB 5: Your code here.

        int r;

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


        //panic("file_block_walk not implemented");
}
```

file_block_walk函数寻找一个文件结构f中的第fileno个块指向的磁盘块编号放入ppdiskbno。



```c
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
       // LAB 5: Your code here.
    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(*ppdiskbno);
    return 0;

    //panic("file_get_block not implemented");
}
```

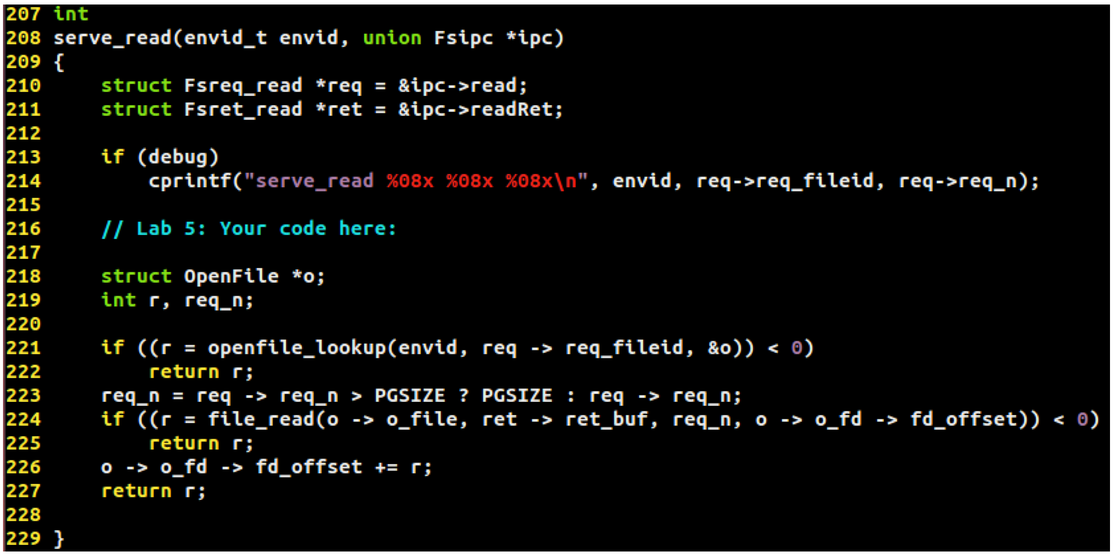
file_get_block函数先调用file_walk_block函数找到文件中的目标块，然后将其转换为地址空间中的地址赋值给blk。



```
  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 (1.1s)
```

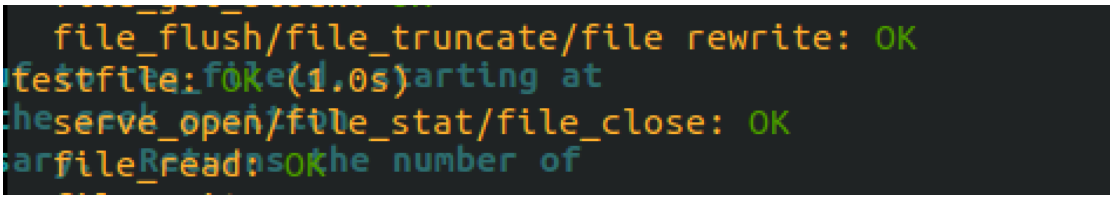
通过测试。

Exercise 5. Implement serve_read in fs/serv.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.



```
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, req_n;

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

}
```

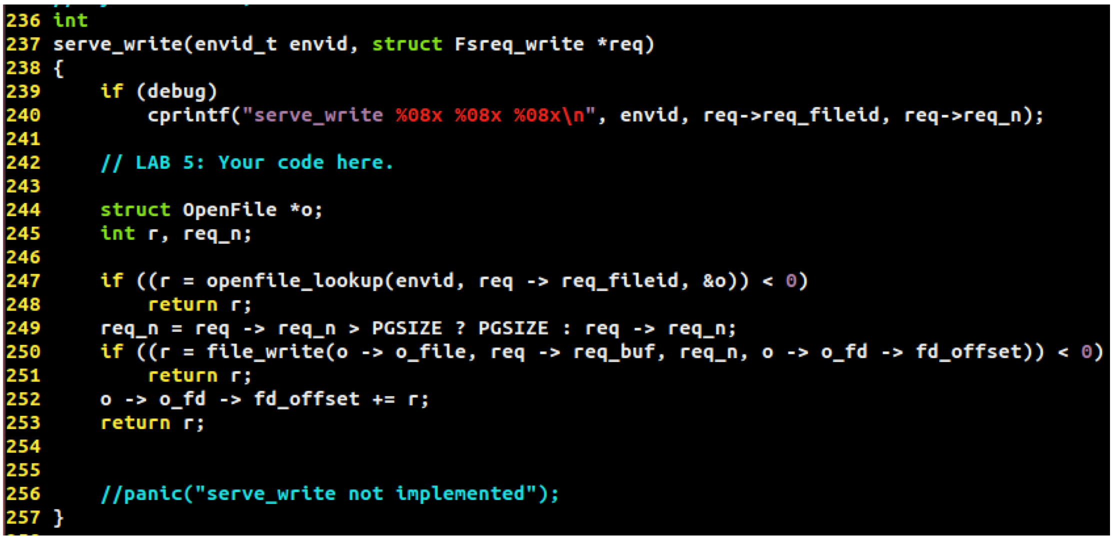
先从Fsipc中获取读请求的结构体，然后在openfile中查找fileid对应的Openfile结构体，紧接着从openfile长相的o_file中读取内容到保存返回结果的ret_buf中，并移动文件偏移指针。



```
  file_flush/file_truncate/file rewrite: OK
testfile: OK (1.0s)
  serve_open/file_stat/file_close: OK
  file_read: OK
```

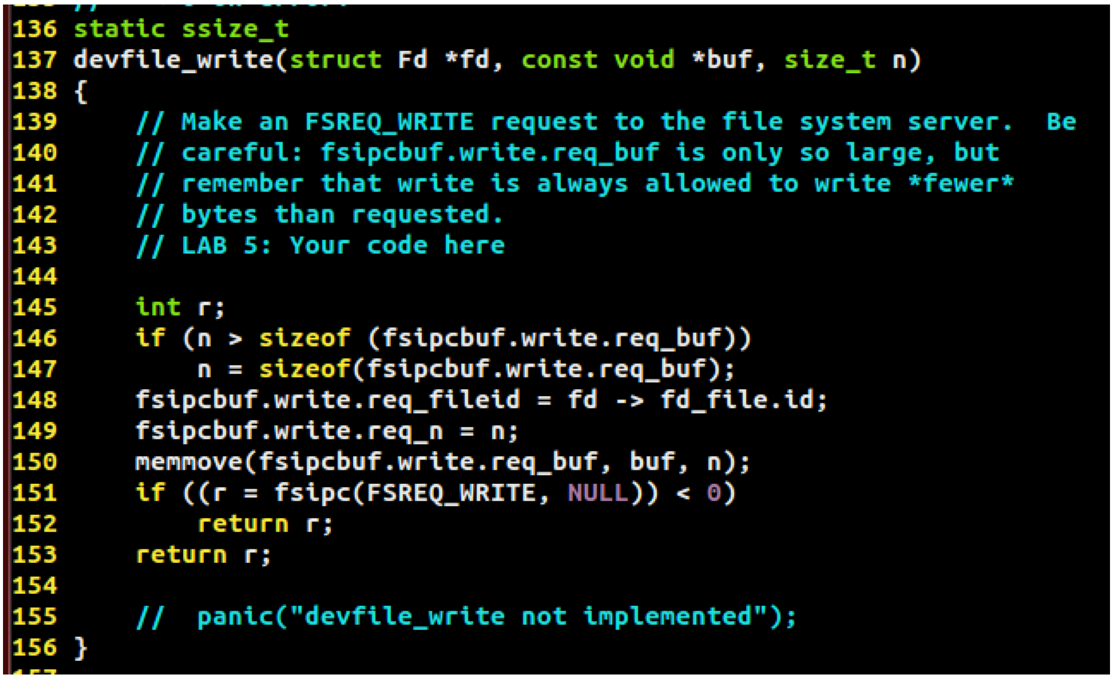
通过测试。

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



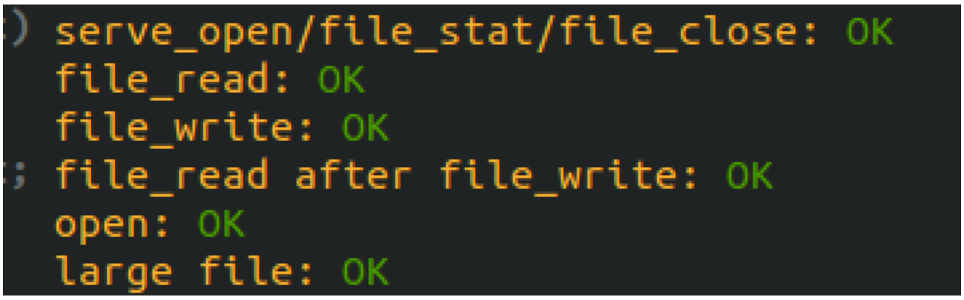
```
int
serve_write(envid_t envid, struct Fsreq_write *req)
{
    if (debug)
        cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);

    // LAB 5: Your code here.

    struct OpenFile *o;
    int r, req_n;

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


    //panic("serve_write not implemented");
}
```

serve_write基本和serve_read是一样的。调用了file_write函数。



```
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

    int r;
    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);
    if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
        return r;
    return r;

    //  panic("devfile_write not implemented");
}
```

devfile_write同样是一个用户库,功能是打包各种参数,然后调用IPC,请求FS内核进程做读取操作。因为是写操作,所以需要 用memmove来把buf里的内容移动到fsipcbuf.write.req_buf.



```
  serve_open/file_stat/file_close: OK
  file_read: OK
  file_write: OK
  file_read after file_write: OK
  open: OK
  large file: OK
```

通过测试。