

## 实验 5 文件系统

班级： 谢志鹏 姓名： 王傲 学号： 15300240004

Q1. 阅读材料，实现在 kern/env.c 文件下的 env\_create 函数，完成磁盘访问。

实现代码段：

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

实现思路及对应代码解释：

实现思路很简单，就是在创建进程（env）时，如果是文件系统的进程，就赋予这个进程 I/O 操作的权限。

对于代码，由于在 kern/init.c 的 i386\_init 函数中，增加了初始化文件系统进程的操作：

```
// Start fs.
ENV_CREATE(fs_fs, ENV_TYPE_FS);
```

将 ENV\_TYPE\_FS（file system）类型赋给了进程创建函数 env\_create，env\_create 检测到创建这个类型的进程时，就赋予这个进程 I/O 操作的权限，即在当前权限的基础上，通过位或操作，增添 FL\_IOPL\_3，而创建其他种类的进程时则不允许赋予这种权限。x86 系统使用 EFLAGS 寄存器中的 IOPL 位来表示是否允许进程执行相关的 I/O 指令。由于要访问的 IDE 硬盘寄存器是在 x86 的 I/O 空间中（而不是 DMA 下的内存中），我们只需要在初始化进程时通过设置 IOPL 位来允许文件系统访问这些寄存器。

这里 JOS 将 IDE 硬盘驱动通过用户态的文件系统进程实现，并且实行的是比较简单的程序驱动 I/O，CPU 通过空转的方式进行轮询，直至磁盘操作完成，而没有通过中断机制实现 I/O。

Q2. 修改 fs/bc.c 中的 bc\_pgfault 和 flush\_block 函数。

bc\_pgfault 函数：

实现代码段：

```
envid_t envid = thisenv->env_id;
void *blkaddr = ROUNDOWN(addr, PGSIZE);

if (sys_page_alloc(envid, blkaddr, PTE_SYSCALL) < 0)
    //allocate a new page for the disk block
    panic("bg_pgfault:can't allocate new page for disk block\n");

if (ide_read(blockno * BLKSECTS, blkaddr, BLKSECTS) < 0)
    //load sectors from the disk to this page
    //BLKSECTS: sectors per block
    panic("bg_pgfault: failed to read disk block\n");

if (sys_page_map(envid, blkaddr, envid, blkaddr, PTE_SYSCALL) < 0)
    //map this page
    panic("bg_pgfault: failed to mark disk page as non dirth\n");
```

### 实现思路及对应代码解释：

bc\_pgfault 函数是一个页错误 (page fault) 处理器。当读取一个块发生页错误时，首先获得虚拟地址，与页的大小 PGSIZE (4K) 对齐，然后进行相应处理：使用 sys\_page\_alloc 函数分配一个块，使用 ide\_read 函数将硬盘上的数据读到这个缓冲区上的刚分配的页（注意，ide\_read 函数是按照扇区读入的），再使用 sys\_page\_map 函数对这个页进行虚拟地址的映射；如果任何一步发生错误，则通过 panic 中断运行。

这里是因为在文件系统中，通过虚拟内存系统实现了一个“缓冲区缓存”：为了将文件系统进程和其他进程的虚拟地址空间孤立开来，JOS 预留了一个 3G 大小的文件系统进程地址空间（从 0x10000000/DISKMAP 到 0xD0000000/DISKMAP+DISKMAX）作为硬盘的内存映射，方便对硬盘上文件的调用。由于文件系统进程和其他进程的虚拟地址空间是独立的，并且文件系统进程唯一需要实现的是文件访问，所以这么做会很方便。

但是，如果将整个硬盘上的内容读进内存的话，会花费很长的时间，所以 JOS 使用“需要时再写入”的方法 (demanding paging)， “假装”硬盘上的内容已经在内存中了。这类似于上次 lab 中实现 fork 时使用的 Copy-On-Write。当需要在这个地址空间上使用某个页，而这个页还没有被从硬盘上读入到内存时，就会触发一个页错误 (page fault)。然后，就需要使用 bc\_pgfault 函数处理页错误：分配一个页，对齐后从硬盘上按照扇区读入，并进行虚拟地址的映射，清空 dirty bit。这样，就完成了从硬盘到内存的页的装载。

flush\_block 函数：

### 实现代码段：

```
addr = ROUNDDOWN(addr, PGSIZE);
//get aligned
int r;
if(va_is_mapped(addr) && va_is_dirty(addr))
//if the block is not in the block cache or is not dirty, does nothing
{
    ide_write(blockno * BLKSECTS, addr, BLKSECTS);
    //write the block back to the disk in sectors
    if((r = sys_page_map(0, addr, 0, addr, PTE_SYSCALL)) < 0)
        //clear the PTE_D bit using sys_page_map
        panic("flush_block: page mapping failed : %e",r);
}
```

### 实现思路及对应代码解释：

这里 flush\_block 函数在必要情况下将缓冲区中的一个块写回到硬盘上。如果这个块不在硬盘的缓冲区上（没有相应的映射关系）或者没有被修改（不是 dirty 的），则不做任何事。

对齐后，利用 va\_is\_mapped 和 va\_is\_dirty 确认是否需要写回，然后使用 ide\_write 函数将这个页按照扇区写回到硬盘上，并使用 sys\_page\_map 函数清空 PTE\_D 位（dirty 位）。

fs/fs.c 文件中 fs\_init 函数说明了这个内存中的硬盘缓冲区的使用方法：

```

// Initialize the file system
void
fs_init(void)
{
    static_assert(sizeof(struct File) == 256);

    // Find a JOS disk. Use the second IDE disk (number 1) if available
    if (ide_probe_disk1())
        ide_set_disk(1);
    else
        ide_set_disk(0);
    bc_init();

    // Set "super" to point to the super block.
    super = diskaddr(1);
    check_super();

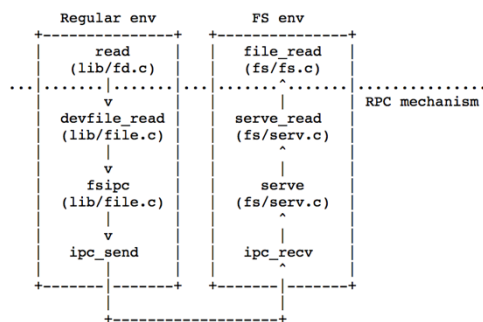
    // Set "bitmap" to the beginning of the first bitmap block.
    bitmap = diskaddr(2);
    check_bitmap();
}

```

当初始化缓冲区后，fs\_init 函数将指向硬盘映射区域的指针存储在全局变量 super 中。所以，我们就可以通过操作 super 结构来操作硬盘上的数据（实际上是内存中的数据，必要时通过页错误处理器读入硬盘上的数据）。这里 super 块是块 1，存储着文件系统的元数据和相关属性。

### Q3. 修改 fs/serv.c，实现 serve\_read，serve\_write 和 lib/file.c 下的 devfile\_write.

JOS 文件系统的使用机制是这样的：



由于其他进程不能直接调用文件系统进程中的函数，所以通过使用基于进程间通信（IPC）的远程进程调用（RPC）实现，类似网络进程通信。从上往下，read 函数可以作用于任何文件描述符（file descriptor），而 read 通过调用 devfile\_read 函数实现对特定的硬盘上的文件进行读取。往下，调用 fsipc 发送 IPC 请求，收到并返回结果。对于服务方，serve 函数中有一个不断从 IPC 接受请求的轮询，将请求分发给正确的处理函数（图中为 serve\_read 函数），并将结果通过 IPC 发回。

JOS 的 IPC 机制允许发送一个 32 位的值，并且（可选的）共享一个页。在文件系统中，这个 32 位的值用来表示请求的类型（类似编号的系统调用），而请求的参数被保存在一个页上的名为 Fsipc 的 union 结构上，通过 IPC 进行共享。在发送方（client），这个页保存在 fsipcbuf 上，而在接收方（server），这个页被映射在 fsreq（0x0ffff000）。

请求方同样通过 IPC 将结果返回。这里，IPC 中传递的 32 位的值用来存放函数的返回码（对于大多数 RPC，这是他们唯一的返回值）。FSREQ\_READ 和 FSREQ\_STAT 函数返回数据，数据被写入发送方共享的页面上。

serve\_read:

实现代码段：

```

struct OpenFile *o;
int r;

```

```

//use openfile_lookup to find the relevant open file
if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;

//req is the request IPC read
//ret is the return IPC readRet

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

//update the seek position
o->o_fd->fd_offset += r;

//return the number of bytes successfully read, or < 0 on error
return r;

```

#### 实现思路及对应代码解释：

主要就是通过 IPC 实现对读文件请求的响应。高负载的读任务通过 file\_read 函数实现，serve\_read 函数只提供了通过 RPC 实现文件读取的接口。

具体实现上，req 为请求方 IPC，ret 为服务方 IPC。通过 openfile\_lookup 函数查找进程标识为 envind 的进程对应的打开文件，然后使用 file\_read 函数，读取当前查询位置 req->req\_fileid 的 req->req\_n 个字节，将读取出来的结果字节返回给请求方的 readRet(ret)，写入 ret\_buf 中，作为返回结果，最后更新查询位置，在文件描述符的 fd\_offset 中。

serve\_write:

#### 实现代码段：

```

struct OpenFile *o;
int r;

//use openfile_lookup to find the relevant open file
if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;

//write req->req_n bytes from req->req_buf to req_fileid,
//starting at the current seek position,
r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset);
if(r < 0)
    return r;

//update the seek position
o->o_fd->fd_offset += r;

```

```

//return the number of bytes written
return r;

```

#### 实现思路及对应代码解释:

实现思路与 serve\_read 十分类似。唯一不同的是响应请求方的请求为写入,调用 file\_write 函数。

具体实现上, req 为请求方 IPC, ret 为服务方 IPC。通过 openfile\_lookup 函数查找进程标识为 envid 的进程对应的打开文件, 然后使用 file\_write 函数, 将 req->req\_fileid 的 req->req\_n 个字节写入当前查询位置, 最后更新查询位置, 在文件描述符的 fd\_offset 中。

devfile\_write:

#### 实现代码段:

```

int r;
//fsipcbuf is a Fsipc union that stores a series of actions including
read, write, etc.

//set the req_fileid of the struct write
fsipcbuf.write.req_fileid = fd->fd_file.id;

//set the number of bytes that are going to write with argument
write.req_n
fsipcbuf.write.req_n = n < PGSIZE ? n: PGSIZE;

//write at most 'n' bytes from 'buf' to 'fd' at the current seek
position
memmove(fsipcbuf.write.req_buf, buf, fsipcbuf.write.req_n);

//make an FSREQ_WRITE request, return the number of bytes successfully
written
r = fsipc(FSREQ_WRITE, NULL);
return r;

```

#### 实现思路及对应代码解释:

devfile\_write 函数和其他 devfile\_\* 函数功能比较类似, 实施文件系统中请求方的的操作, 将请求方的请求参数打包在 Fsipc 这个 union 结构体中, 放在 fsipcbuf 上, 通过共享页与接收方共享。然后, 调用 fsipc 函数通过 IPC 发送请求。

union Fsipc 包括内容如下:

```

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;

    // Ensure Fsipc is one page
    char _pad[PGSIZE];
};

```

具体实现上，`devfile_write` 函数将 `Fsipc` 设置为 `write (Fsreq_write)`，设置 `req_fileid` 参数和要读取的字节数 `req_n`，然后通过 `memmove` 函数将 `buf` 的 `n` 个字节写到当前位置的 `req_buf`，最后调用 `fsipc` 函数，传递 `FSREQ_WRITE`，发起 IPC 请求，返回写入的字节数。

**Q4. 阅读材料，在 `lib/spawn.c` 中修改 `copy_shared_pages` 共享文件描述符状态。**

实现代码段：

```

uintptr_t i;
for (i = 0; i < USTACKTOP; i += PGSIZE)
    //loop through all page table entries in the current process
    {
        if ((uvpd[PDX(i)] & PTE_P) && (uvpt[PGNUM(i)] & PTE_P) &&
            (uvpt[PGNUM(i)] & PTE_SHARE))
            //search any page mappings that
            //have the PTE_SHARE bit set into the child process
            {
                sys_page_map(0, (void*)i, child, (void*)i, (uvpt[PGNUM(i)] &
                    PTE_SYSCALL));
                //copy the mappings for shared pages into the child address space
            }
    }
return 0;

```

实现思路及对应代码解释：

`copy_shared_pages` 所做的工作就是遍历当前进程（父进程）的所有页表项，找出所有 `PTE_SHARE` 位被设置的页面映射（需要共享的页），将共享页面的映射拷贝到子进程的地址空间中。

具体的代码实现上，遍历 `USTACKTOP` 栈顶以下的页面，寻找 `PTE_SHARE` 位被标记的页表项（需要共享），然后利用 `sys_page_map` 函数将这个映射拷贝到子进程的地址空间中。

这么做的原因是，我们希望在父进程和通过 fork 或 spawn 生成的子进程之间共享文件描述符状态（file descriptor state），但文件描述符状态是保存在用户内存空间中的页上的。在 fork 中，内存上的页是 COW（Copy On Write）的，需要使用时状态会被复制而不是共享；而对于 spawn，则根本不会复制父进程中内存中的页。为了解决这个问题，定义了 PTE\_SHARE 位，对于 fork 和 spawn，一旦父进程中检测到这一位被设置了的页表项，这个页表项就被视为需要共享，需要被从父进程拷贝到子进程。

实现结果：

```
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
PCI: 00:02.0: 1234:1111: class: 3.0 (Display controller) irq: 0
PCI: 00:03.0: 8086:100e: class: 2.0 (Network controller) irq: 11
PCI function 00:03.0 (8086:100e) enabled
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good
fork handles PTE_SHARE right
spawn handles PTE_SHARE right
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

```
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
PCI: 00:02.0: 1234:1111: class: 3.0 (Display controller) irq: 0
PCI: 00:03.0: 8086:100e: class: 2.0 (Network controller) irq: 11
PCI function 00:03.0 (8086:100e) enabled
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good
going to read in child (might page fault if your sharing is buggy)
read in child succeeded
read in parent succeeded
Welcome to the JOS kernel monitor!
```

Q5. 阅读材料，在 kern/trap.c 中调用 kbd\_intr 来处理 IRQ\_OFFSET+IRQ\_KBD 中断和调用 serial\_intr 来处理 IRQ\_OFFSET+IRQ\_SERIAL 中断。

实现代码段：

```
//handle keyboard interrupt
if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD)
{
    kbd_intr();
    return;
}
```



```

//handle serial interrupt
if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL)
{
    serial_intr();
    return;
}

```

### 实现思路及对应代码解释：

实现思路很简单，根据不同的中断类型（kbd 或者 serial），调用相应的处理函数来处理中断。

为了让 shell 能够工作，我们需要能够通过打字输入。对于 QEMU，在图形窗口通过打字的输入对于 JOS 来说就是从键盘的输入，对控制台(console)的输入对于 JOS 来说就是串口(serial port)的输入。我们需要处理这两种 I/O 中断。trap IRQ\_OFFSET+IRQ\_KBD 表示键盘 (keyboard) 输入带来的中断，需要调用 kbd\_intr 处理；trap IRQ\_OFFSET+IRQ\_SERIAL 表示串口带来的中断，需要调用 serial\_intr 处理。

实验结果：

```

check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
PCI: 00:02.0: 1234:1111: class: 3.0 (Display controller) irq: 0
PCI: 00:03.0: 8086:100e: class: 2.0 (Network controller) irq: 11
PCI function 00:03.0 (8086:100e) enabled
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good
Type a line: ns: 52:54:00:12:34:56 bound to static IP 10.0.2.15
NS: TCP/IP initialized.
Hello World!
Hello World!
Type a line: This is the shell input test.
This is the shell input test.

```

使用一些基本的 shell 指令：

```

$ echo hello world | cat
hello world
$ cat lorem | cat
lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim
id est laborum.
$ cat lorem | num7777777777777777
1 Lorem ipsum dolor sit amet, consectetur
2 adipiscing elit, sed do eiusmod tempor
3 incididunt ut labore et dolore magna
4 aliqua. Ut enim ad minim veniam, quis
5 nostrud exercitation ullamco laboris
6 nisi ut aliquip ex ea commodo consequat.
7 Duis aute irure dolor in reprehenderit
8 in voluptate velit esse cillum dolore eu
9 fugiat nulla pariatur. Excepteur sint
10 occaecat cupidatat non proident, sunt in
11 culpa qui officia deserunt mollit anim
12 id est laborum.
$ cat lorem | num | num | num
1 1 1 Lorem ipsum dolor sit amet, consectetur
2 2 2 adipiscing elit, sed do eiusmod tempor
3 3 3 incididunt ut labore et dolore magna
4 4 4 aliqua. Ut enim ad minim veniam, quis
5 5 5 nostrud exercitation ullamco laboris
6 6 6 nisi ut aliquip ex ea commodo consequat.
7 7 7 Duis aute irure dolor in reprehenderit
8 8 8 in voluptate velit esse cillum dolore eu
9 9 9 fugiat nulla pariatur. Excepteur sint
10 10 10 occaecat cupidatat non proident, sunt in
11 11 11 culpa qui officia deserunt mollit anim
12 12 12 id est laborum.
$ lsfd
fd 0: name <cons> lsdtr 0 size 0 dev cons
fd 1: name <cons> lsdtr 0 size 0 dev cons

```



Q6. 阅读材料，在 user/sh.c 中实现 shell 支持 IO 重定向。

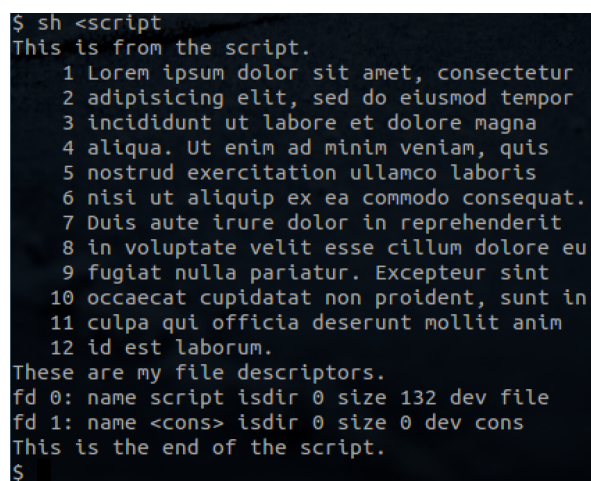
实现代码段：

```
if ((fd = open(t, O_RDONLY)) < 0)
//open 't' for reading as file descriptor 0 as standard input
//we can't open a file onto a particular descriptor,
//so open the file as 'fd'
{
    cprintf("open %s for read: %e", t, fd);
    exit();
}
if (fd != 0)
//check whether 'fd' is 0
{
    //if not, dup 'fd' onto file descriptor 0
    dup(fd, 0);
    //then close the original 'fd'
    close(fd);
}
break;
```

实现思路及对应代码解释：

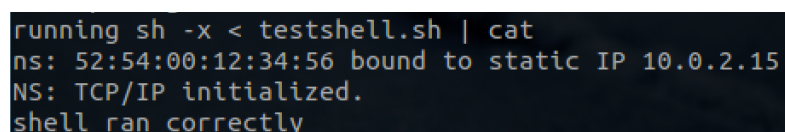
运行 icode 进程，它先会运行 init 进程，将 console 作为输入输出文件描述符，然后运行 spawn 脚本，即 shell。我们需要实现 shell 支持 IO 重定向。将 t 以只读方式（read only）打开，作为文件描述符 0（标准输入）。由于不能指定特定的文件描述符，所以打开为 fd。如果 fd 不为 0，就将 fd 复制到文件描述符 0，然后关闭原始 fd。

使用 sh <script 指令结果：



```
$ sh <script
This is from the script.
 1 Lorem ipsum dolor sit amet, consectetur
 2 adipiscing elit, sed do eiusmod tempor
 3 incididunt ut labore et dolore magna
 4 aliqua. Ut enim ad minim veniam, quis
 5 nostrud exercitation ullamco laboris
 6 nisi ut aliquip ex ea commodo consequat.
 7 Duis aute irure dolor in reprehenderit
 8 in voluptate velit esse cillum dolore eu
 9 fugiat nulla pariatur. Excepteur sint
10 occaecat cupidatat non proident, sunt in
11 culpa qui officia deserunt mollit anim
12 id est laborum.
These are my file descriptors.
fd 0: name script isdir 0 size 132 dev file
fd 1: name <cons> isdir 0 size 0 dev cons
This is the end of the script.
$
```

使用 make run-testshell 指令结果：



```
running sh -x < testshell.sh | cat
ns: 52:54:00:12:34:56 bound to static IP 10.0.2.15
NS: TCP/IP initialized.
shell ran correctly
```

实验结果贴图（全部代码完成后，make grade）：

```
internal FS tests [fs/test.c]: Timeout! OK (31.2s)
fs i/o: OK
check_bc: OK
check_super: OK
check_bitmap: OK
testfile: Timeout! OK (30.4s)
serve_open/file_stat/file_close: OK
file_read: OK
file_write: OK
file_read after file_write: OK
open: OK
large file: OK
PTE_SHARE [testpteshare]: OK (1.0s)
PTE_SHARE [testfdsharing]: OK (1.1s)
start the shell [icode]: OK (0.9s)
testshell: OK (2.4s)
(Old jos.out.testshell failure log removed)
primespipe: OK (6.7s)
Score: 115/115
```

### 实验感想及难点：

相对以前的实验来说，本次文件系统的实验不是很难。不过本次实验使用了很多以前实验实现的机制，比如 IPC，fork 等等，需要对以前的内容掌握的比较好，做起来才不会十分困难。

比较困难的地方在于新加入的文件系统机制带来了新的结构和使用方法，同时需要修改一部分旧有的机制，需要比较熟悉才能按照指导正确完成相应功能。

JOS 文件系统的一个比较有趣的地方是底层是通过 IPC 实现的，很类似网络进程通信。考虑到 JOS Lab 6 网络部分我们不会去做，还是比较遗憾的。

这是我们操作系统的最后一个 Lab，完成后真的很有成就感，尤其是最终实现的 shell，可以像正常的 Unix/Linux 操作系统一样使用 ls、echo、cat 等常用指令，有了可以投入实际使用的雏形，真的很令人高兴。这也是为操作系统的 Lab 画上了一个圆满的句号。

JOS 的设计真的是十分优秀，可以看出其设计者们深厚的 C 语言和操作系统功底。事实证明，仅仅课堂上讲的知识是远远不够的，必须通过实践才能巩固，获得新的知识。这五次 lab，每次都要花费几天时间才能完成，做起来很累，很辛苦，但也很值得。这使得我真正的从底层了解了操作系统是如何“跑起来”的，如何实现各种各样的功能。完成这样的 lab，操作系统的课程才是圆满的。6.828，这个数字，我这辈子大概都不会忘记了。