操作系统及实习（实验班）

JOS-Lab5
文件系统 & Spawning & Shell
实习报告

姓名 杜若谷 学号 1300012855
日期 2016.6.1

目录

内容一：总体概述.....3
内容二：任务完成情况.....3
任务完成列表（Y/N）.....3
具体 Exercise 的完成情况.....3
内容三：遇到的困难以及解决方法.....17
内容四：收获及感想.....177
内容五：对课程的意见和建议.....17
内容六：参考文献.....17

内容一：总体概述

这个 Lab 中，我们首先需要完成一个文件系统。这个文件系统部分在内核内，而是在内核外以有使用特殊 I/O 指令权限的进程来实现的。我们首先需要研究已经完成的代码，实现文件系统内部的代吗。然后完成远程过程调用(RPC)的机制，让其他用户进程可以使用文件系统。

随后我们需要完成 Spawn 机制。其过程类似于 fork()后执行 exec()，但是由于实现的困难性，我们目前不采用 exec()，而是采用 Spawn 的机制，即父进程直接设置子进程运行新程序。

最后我们要实现 Shell 的一些功能。在此之前我们还需要能够支持在用户进程内使用键盘进行输入。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise 1	Exercise 2	Exercise 3	Exercise 4	Exercise 5	Exercise 6
Y	Y	Y	Y	Y	Y
Exercise 7	Exercise 8	Exercise 9	Exercise 10		Challenge 2
Y	Y	Y	Y		Y

具体 Exercise 的完成情况

第一部分

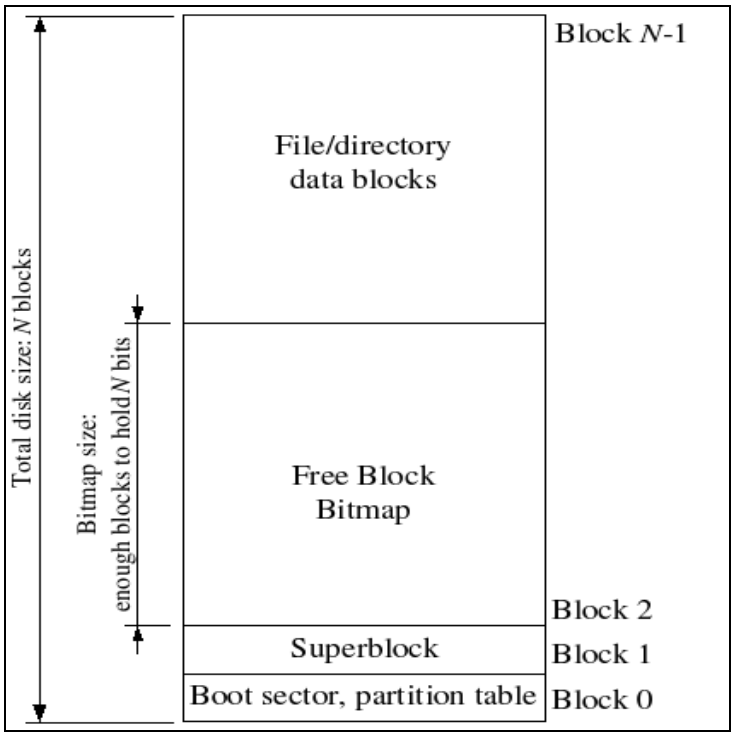
开始这一部分之前，需要测试之前的样例程序是否能够执行。在初始的测试中出现了一些问题。经过检查发现根本没有加载测试程序。检查后发现问题出在 env_create()函数中在 Lab 5 中去掉了 load_icode()函数，这本可以理解，因为在这个 Lab 中需要对 env_create()进行修改。但是如果目前去掉这一部分还做之前的测试明显就不能通过了。我认为这有可能是远程 Git 仓库中 Lab 5 的 branch 没有设置好的地方。

测试完成后，在去掉之前做的一些注释后就可以进入第一部分了。首先来介绍一下在这个部分中需要完成的文件系统的架构。

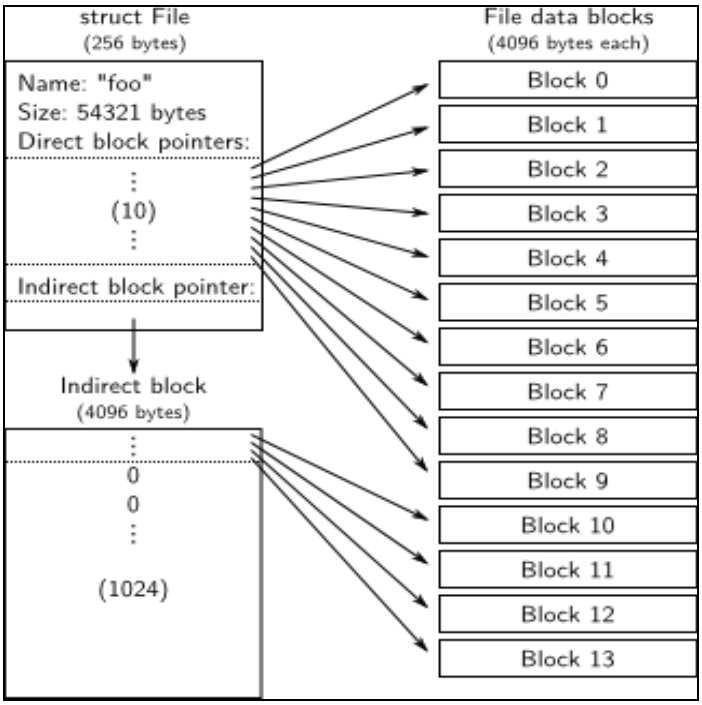
很多系统中，文件系统的元数据需要特定的操作系统接口才能读取，这是为了能够方便的修改文件系统内部结构而保持接口不便。而 JOS 的文件系统的元数据能够直接被用户程序读取，因此我们可以设计用户态的目录索引工具 ls。我们仍然需要修改部分内核以便我们的程序能够获得特殊的权限。

在目前的系统中，所虚拟出来的磁盘的存储单位是 Sector,大小为 512KB，在读写时应当以 Sector 为单位。JOS 操作系统中管理磁盘的单位为 Block，设定

为 4096KB，正好和页的大小一致。在磁盘的 0 号 Block 中存储了 Boot Loader，在 1 号 Block 中存储了我们的 Superblock。在 Superblock 中我们存储了整个文件系统的元数据，例如 Block 的大小，整个磁盘的大小，根目录有关信息，整个系统最后一次被挂载的时间，整个系统最后一次出错的时间，等等。很多文件系统复制了很多份 Superblock 分散在整个系统的各个位置，防止某一部分被损坏造成 Superblock 的丢失。在 JOS 中 Superblock 的结构由 inc/fs.h 中的 struct Super 结构体来描述。



一般 UNIX 系统的文件系统都会采用 i-node 和目录共存的结构，而我们要实现的系统中为了实现简便，不需要实现 i-node,而是直接在目录文件中保存了子文件或者子目录的元数据。在 JOS 的文件系统中，目录和普通文件都是作为文件来管理，它们的元数据都由 inc/fs.h 的 struct File 结构来描述，因此目录文件中保存的子目录或这文件的元数据就是一个个 struct File 结构体。目录和普通文件不同的是操作系统需要理解和管理目录文件的信息。



在一个 struct File 结构体中，记录了文件的名称、大小和种类（目录还是文件）等信息，还有储存文件内容的 Block 的编号。首先在 f_direct 数组中存储了 10 个直接索引的编号，即储存文件内容的前十个块都可以由这个数组直接得到。除此之外 File 结构体中还有一个间接索引的 Block 编号。这个 Block 中存储的是 1024 个额外的 Block 编号。因此在 JOS 的文件系统中，最大的文件大小为 1034 个 Block，即 4MB+40KB。

Exercise 1

我们在这个 Exercise 中需要实现对硬盘的访问。与一般系统在内核中实现一个 IDE 硬盘驱动不同，我们的 JOS 中则是使用一个用户态的系统程序来实现对硬盘的读写。并且修改内核使其有必需的权限来访问硬盘。

在用户态程序中我们可以使用轮询的方式来访问硬盘。我们没有使用中断驱动的原因是需要修改内核来实现存储硬件发出的中断，随后分发到合适的用户程序的过程。

在 x86 体系结构中使用 EFLAGS 中的 IOPL 位来决定在保护模式下是否允许使用特殊 I/O 指令，比如 IN 和 OUT。由于我们需要直接使用这些指令来读写硬盘而不是使用内存映射的方法，因此我们需要设置 EFLAGS 来让用户程序有权限使用这些指令，但是我们只希望和文件系统有关的用户程序而非普通用户程序能够使用 I/O 指令。

在实际实现中，需要实现对不同用户程序不同的待遇非常简单，因为在上下文切换的时候，所有寄存器的值会被上 CPU 进程的 Env 中的 Trapframe 中的数据替换，并且被储存在自己的 Env 的 Trapframe 中。因此我们只需要在创建系统进程的时候把 Env 中 Trapframe 的 EFLAGS 的 IOPL 位设置好就可以实现让文件系统进程能够使用 I/O 指令了。

```
e->env_type = type;
if(e->env_type == ENV_TYPE_FS) {
    e->env_tf.tf_eflags &= ~FL_IOPL_MASK;
    e->env_tf.tf_eflags |= FL_IOPL_3;
}
load_icode(e, binary);
```

完成这一部分后成功通过 make grade 的 fs i/o 测试。

Question 1

我们除此之外不用在做其他工作来保证文件系统进程的 I/O 权限了。原因其实在之前已经叙述的十分清楚，如果了解 JOS 的运行过程，就可以知道在上下文切换的过程中保存现场和恢复现场都会涉及 EFLAGS 的内容，因此只要在创建进程的时候把 EFLAGS 中的 IOPL 位设置好并且在之后不再修改，就能保证这些进程在运行时能够一直有权限使用 I/O 指令了。

保存现场的代码如下在 kern/trap.c 的 trap()函数中，Trapframe 中已经包括 EFLAGS。

```
// Copy trap frame (which is currently on the stack)
// into 'curenv->env_tf', so that running the environment
// will restart at the trap point.
curenv->env_tf = *tf;
// The trapframe on the stack should be ignored from here on.
tf = &curenv->env_tf;
```

恢复现场的代码如下。POPAL 指令已经把 EFLAGS 恢复了。

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

    __asm __volatile("movl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

Exercise 2

在这一个 Exercise 中，我们需要完成文件系统的块缓冲机制，即使用内存缓冲对于磁盘的读写。这一部分的实现在 fs/bc.c 中。

在 JOS 中，块缓冲的实现十分的原始。这里的文件系统最多能够维护的磁盘大小为 3GB,这是因为在文件系统的用户进程中只预留了 3GB 作为磁盘缓冲区(从 0x10000000 到 0xD0000000),而我们需要将整个磁盘映射到这 3GB 的内存空间上。

由于现在实际的磁盘的空间都远远大于 3GB，因此 JOS 的实现就显得非常不实用。但是在 64 位的系统中由于目前磁盘空间距离占满地址空间还有很大的距离，因此这样实现块缓冲可能是可行的。

在具体实现中，我们并非需要将整个磁盘读入内存，只需像 fork()函数一项使用用户缺页处理程序使用按需调页即可。因此我们首先需要实现 fs/bc.c 中的 bc_fault()用户缺页处理程序。

我们可以研究一下 bc_fault()函数，首先它对出现缺页的地址进行了越界检查，确认是我们需要的缓冲区的页后，我们就需要从磁盘读入了。之后我们可以仔细分析一下后半段程序。

```
if((r = sys_page_alloc(0, ROUNDDOWN(addr, PGSIZE), PTE_W | PTE_U) < 0))
    panic("bc_pgfault: page alloc: %e", r);

r = ide_read(
    blockno * BLKSECTS,
    ROUNDDOWN(addr, BLKSIZE),
    BLKSECTS
    );
if(r < 0) panic("bc_pgfault: ide read %e", r);

// Clear the dirty bit for the disk block page since we just read the
// block from disk
if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] & PTE_SYSCALL)) < 0)
    panic("in bc_pgfault, sys_page_map: %e", r);

// Check that the block we read was allocated. (exercise for
// the reader: why do we do this *after* reading the block
// in?)
if (bitmap && block_is_free(blockno))
    panic("reading free block %08x\n", blockno);
```

从磁盘读入使用了 fs/ide.c 中的 ide_read()函数。之后值得注意的是我们需要清空这一页上的 Dirty 位。因为我们刚刚对新分配的页进行了写入，因此此时 Dirty 位应当被设置。但是我们还需要根据这一位判断是否需要写回磁盘，在这时明显缓存内容和磁盘没有区别，因此我们应当清楚 Dirty 位。在清除中使用的方法则是使用 sys_page_map()重新挂载这一页。这时我们需要重新查看一下系统调用，尤其是 page_insert()中对这种情况的处理办法有没有问题。

之后才来判断这一个 Block 是否是空白的，如果是的话说明出错。需要做完读磁盘之后再做这一步是有原因的。如下图，block_is_free()需要使用 bitmap。但是 bitmap 也是从磁盘上映射过来，如果放在读取磁盘之前检查则可能会造成缺页处理程序反复调用，永远无法跳出迭代。

```
bool
block_is_free(uint32_t blockno)
{
    if (super == 0 || blockno >= super->s_nblocks)
        return 0;
    if (bitmap[blockno / 32] & (1 << (blockno % 32)))
        return 1;
    return 0;
}
```

在研究完 bc_fault()后，我们再看一下 flush_block()函数。这个函数首先检查 addr 是否是属于缓冲区。随后查看这一区域是否有页面和是否有修改。如果没有页面或者没有修改则表明不用写回磁盘，函数直接返回。之后就会把页面写回磁盘，并且清空 Dirty 位。需要注意的是写回磁盘时和重新挂载这一页来清空 Dirty 位时需要注意需要将 addr 取整到页首。

```
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");
void *r_addr = ROUNDDOWN(addr, BLKSIZE);
int r;

if(!va_is_mapped(addr))
    return;
if(!va_is_dirty(addr))
    return;

r = ide_write(
    blockno * BLKSECTS,
    r_addr,
    BLKSECTS);
if(r < 0) panic("flush_block: ide write %e", r);

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

这一部分完成后成功通过了 make grade 的 check_bc、check_super 和 check_bitmap 测试。

Exercise 3

这一节需要维护 bitmap 的功能。由于我们之前已经设置好磁盘的缓冲区，因此从磁盘 2 号 Block 开始的 bitmap 可以当作内存中的一个数组来使用。当找到一个空闲块后应当将对应地 bitmap 位设置为 0，并且立刻 flush 对应地 bitmap 所在 Block，因为如果在 bitmap 写回磁盘之前系统崩溃，就会造成已经分配的块的 bitmap 还是显示为空闲块，造成文件系统不一致，可能会造成风险。因此我们需要在这里使用刚刚完成的 flush_block()函数。

```
int i;
for(i = 1; i < DISKSIZE/BLKSIZE; i++) {
    if(block_is_free(i)) {
        bitmap[i/32] &= ~(1<<(i%32));
        flush_block(&bitmap[i/32]);
        return i;
    }
}
```

完成以上工作后成功通过 make grade 的 alloc_block 测试。

Exercise 4

文件 inc/fs.c 中的函数是用来维护 struct File 结构体的。Write up 要求我们了解这个文件中的所有函数。

首先我们来看初始化整个文件系统的 fs_init()函数。这个函数中首先检查除了第 0 号磁盘外是否还有第 1 号磁盘。在 Lab 5 中我们应当能够检查到这一块磁盘，所以之后就会调用 IDE 磁盘驱动从启动盘切换到编号为 1 的磁盘。之后这个函数会初始化 Block cache，设置 Super block 和 bitmap 所对应的内存地址，

并且做相应检查。

我们需要完成的 file_block_walk()函数的功能是给定文件的 struct File 结构体和文件内块编号，给出这一块在磁盘内实际编号（的指针），同时在特定情况下有可能会为文件分配非直接引用 Block。值得注意的是需要分配新 Block 的时候，需要对新的 Block 清空。因为我们是以块号是否为 0 判断引用是否有效，新分配的块明显没有任何引用，因此应当全部为 0.

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

之后我们需要实现的函数是 file_get_block()，这个函数基本上是对 file_block_walk()的包装，不同的是这个函数需要返回指定文件内的第 filebno 块所在的内存地址。因此如果原本文件内还没有第 filebno 块，我们就需要为这个文件分配一个磁盘块并且将这一块的编号加到 struct File 结构体中和这间接引用块的指定位置。

之后是 dir_look_up()函数，这个函数的功能非常简单，就是给定目录文件的 File 结构体，在其中寻找指定文件名的文件，如果找到了则返回这个文件的 File 结构体。

我们接着研究 dir_alloc_file()函数，这个函数的功能是给定目录文件的 File 结构体，在这个结构体中新分配一个子文件的 File 结构体并且返回。这个函数并不负责新的 File 结构体内容的填充。这个函数首先在目录文件内遍历所有已有的块，并且在每个块内使用 file_get_block()查找有没有空闲位置可以存放 File 结构体，如果找到就可以返回。如果都没有空闲位置了就使用 file_get_block()新分配一个块，并且修改目录文件的大小，返回这个块的首地址就可以了。从在块内寻找空闲未知的代码可以看出，JOS 的文件系统是使用文件名是否为空串来判断是否存在 File 结构体。说明任何文件的文件名不能为空串。

```
for (i = 0; i < nblock; i++) {
    if ((r = file_get_block(dir, i, &blk)) < 0)
        return r;
    f = (struct File*) blk;
    for (j = 0; j < BLKFILES; j++)
        if (f[j].f_name[0] == '\0') {
            *file = &f[j];
            return 0;
        }
}
```

之后我们来看一下 walk_path()函数，这个函数的功能是根据指定地址找到所对应的文件及其所在的目录。如果没有找到文件且已经遍历到最后一层，这个函数还会在 lastelem 中存储最后一个找到的文件名，并且在 pdir 中存储目录的 File 结构体。

函数 file_create()调用了 walk_path()功能。这个函数需要给定的目录能够找到最后一层，但是不能找到对应文件。函数 file_open()也使用了 walk_path()，具体不再赘述。

函数 file_read()功能是从指定给定文件的 File 结构体，从这个文件中读取从 offset 开始的 count 字节。函数 file_write()也类似。这两个函数都调用了 file_get_block()

file_free_block()的功能是从指定 File 结构体和文件内块号 filebno，释放文件中的这一块。

函数 file_truncate_blocks()的功能是将一个文件的块数减小，但是不修改 File 中的文件大小。这个函数的作用是供 file_set_size()调用，如果缩小文件的大小，就会调用之前的函数缩减块数。随后调用 flush_block()。

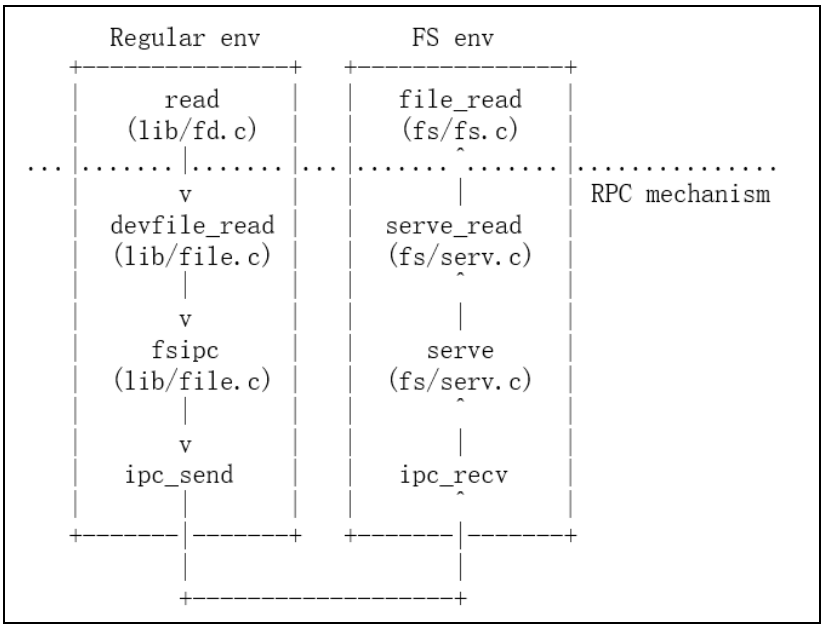
函数 file_flush()的功能是把一个文件的元数据(File 结构体)和里面所有的内存块写回磁盘。文件中可能部分块未分配，这个函数会跳过这些块。

函数 fs_sync()的功能是把所有的 Block cache 写回磁盘。

Exercise 5

完成之前的部分，我们就已经实现了进程内部的文件系统管理程序。但是为了让外部程序能够使用文件系统，我们应当为其添加相关的接口。而非常明显，任何进程对文件的使用都应当会对我们实现的文件系统管理进程进行访问，这就是个很典型的进程间通信问题。而目前我们已经实现了一套简单的 IPC 系统，因此我们将使用这一套系统实现普通进程对文件系统的使用。

JOS 将这一套使用 IPC 对其他进程中过程的调用称为 Remote Procedure Call(RPC)，这一套系统的基本架构如下：



在虚线以下的部分被称作 RPC 机制。我们从本质来看，左半部分是普通进程内部的用户调用，而右半部份则是文件系统进程内部的调用，两个部分中间的连线则是 IPC 机制。通过以上这样一个系统，我们就可以将内核外的文件系统开放给其他进程使用。

接下来我们具体分析这一套系统的工作流程。首先在 lib/fd.c 中实现了一套与 UNIX 相同的文件读写函数。这些函数一般都是根据指定 fd 编号，对文件进行操作，这些函数包括 close(), dup(), read(), readn(), write(), seek(), stat()等。 但是 open()函数的实现确是在 lib/file.c。所以我并不是很理解这

两个文件中的函数究竟是根据什么标准分类的。

应当是用户程序一般使用的，在这些函数之下就是一些可能涉及 struct Fd 结构体的函数。因为在之前的函数中对文件操作的标识符只有 fd 编号。在这些涉及 Fd 结构体的操作中，有些并不需要使用 IPC 操作。因为所有有关的 Fd 结构体都已经被挂载到本地，每个 Fd 结构体占一页，被挂载到了从 FDTABLE 开始的虚拟地址，FDTABLE 位置的 Fd 编号为 0，以此类推。通过查询这些 Fd 结构体，就可以获得 dev_id，offset 和文件 id，模式等信息。

我们需要通过 Dev 作为媒介才能使用 devfile_read()等函数。每个 Dev 对应一种文件类型，在 JOS 中目前只有三种 dev：devfile，devpipe，devcons。每种 Dev 有不同的 dev_id，dev_name，dev_read()，dev_close()，dev_stat()，dev_write()，dev_trunc。通过 Fd 中的 dev_id，我们可以得到这个文件所在的 Dev，再根据其中的函数指针就能够找到相应操作的函数，例如 devfile_read()。

```
struct Dev devfile =
{
    .dev_id =    'f',
    .dev_name = "file",
    .dev_read = devfile_read,
    .dev_close =    devfile_flush,
    .dev_stat = devfile_stat,
    .dev_write =    devfile_write,
    .dev_trunc =    devfile_trunc
};
```

```
static struct Dev *devtab[] =
{
    &devfile,
    &devpipe,
    &devcons,
    0
};

int
dev_lookup(int dev_id, struct Dev **dev)
{
    int i;
    for (i = 0; devtab[i]; i++)
        if (devtab[i]->dev_id == dev_id) {
            *dev = devtab[i];
            return 0;
        }
    cprintf("[%08x] unknown device type %d\n", thisenv->env_id, dev_id);
    *dev = 0;
    return -E_INVAL;
}
```

调用通过 Dev 找到的函数后，IPC 函数就会被调用（fsipc()），之后的工作便交给文件系统管理程序。在 fs/serv.c 中维护了一个记录当前打开文件的数组，每个打开的文件有一个 OpenFile 结构体。这个结构体把每个文件的 Fd（需要共享给其他程序）和 File（文件系统自己保存）联系起来。在初始化阶段为每个打开的程序分配一个 o_fileid，其值等于 OpenFile 结构体在 opentab 数组中的 index，然后给 OpenFile 中的 Fd 指定一个一页大小的空间，以便未来共享给用户程序。

比较巧妙的是，文件系统查看某个 OpenFile 是否被分配使用的方法是查看

Fd 所在物理页的引用数，如果小于等于 1 则说明没有被打开。这样做就可以方便用户程序对文件的关闭操作，而且在进程退出时也不用显示通知文件系统管理程序。

```
int
openfile_lookup(envid_t envid, uint32_t fileid, struct OpenFile **po)
{
    struct OpenFile *o;

    o = &opentab[fileid % MAXOPEN];
    if (pageref(o->o_fd) <= 1 || o->o_fileid != fileid)
        return -E_INVAL;
    *po = o;
    return 0;
}
```

之后的过程就是 serve()函数反复调用 ipc_recv()后接收到请求，将发送的页 map 到 fsreq 的位置，并且根据所发送的请求种类编号分发到相应的 handler，随后再使用 ipc_send()将结果返回，随后 unmount fsreq 位置的页。具体内容就不再赘述了。

在这个 Exercise 中，我们需要完成的是 fs/serve.c 的 serve_read()。根据以上的分析，我们可以看出这个函数是位于文件系统管理程序的针对读文件的 handler。这个部分需要注意的是从文件读取内容的大小不能超过返回 buffer 的大小，即 PGSIZE。在读取完成后，还需要把 Fd 的 offset 加上读取的字节数。

```
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;
    char *read_buf = ret->ret_buf;

    if((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
        return r;
    size_t readsize = MIN(PGSIZE - 1, req->req_n);
    read_buf[PGSIZE - 1] = '\0';

    ssize_t s_read;
    if((s_read = file_read(o->o_file, read_buf, readsize, o->o_fd->fd_offset)) < 0)
        return s_read;

    o->o_fd->fd_offset += s_read;

    return s_read;
}
```

Exercise 6

在这一个 Exercise 中我们需要完成的是 fs/serve.c 的 serve_write()和 lib/file.c 的 devfile_write。

函数 serve_write()和 serve_read()区别并不大，我们就不再赘述了。接下来我们研究一下 devfile_write()函数。可以看出，这个函数位于用户程序部分，

作用就是根据 Fd 的 Dev 所指定的 write 函数，处理写入请求。需要注意的是，由于 IPC 传递的 buffer 大小有限，因此需要预先检查需要写入的字节数是否越界。

```
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
    //panic("devfile_write not implemented");
    int r;
    ssize_t write_size;

    fsipcbuf.write.req_fileid = fd->fd_file.id;
    fsipcbuf.write.req_n = MIN(n, PGSIZE - (sizeof(int) + sizeof(size_t)));
    memcpy(fsipcbuf.write.req_buf, buf, fsipcbuf.write.req_n);

    r = fsipc(FSREQ_WRITE, NULL);
    return r;
}
```

完成以上工作后，可以成功通过 make grade 的第一部分的 Test.

第二部分

这一部分的工作是完善 spawn 以及有关的函数。这个函数的功能类似与在 UNIX 中运行 fork()后再运行 exec()，但是在 JOS 中我们选择目前不使用 exec()而是 spawn()。我认为这样做的原因是 exec()需要在自己的内存地址内实现读取 ELF 文件，还需要重新设置栈，这些工作很难在不影响现在程序状态的情况下完成，只有设置寄存器可以通过即将完成的 sys_set_trapframe()系统调用来实现，其他很多功能都需要系统调用来完成。

Spawn()的实现和之前的 load_icode()十分相似。首先从指定地址读取 ELF 头，检查 magic 值之后证明已经打开一个 ELF 文件。随后使用 sys_exofork()创建子进程。调用 init_stack()为子进程设置好栈，随后使用 map_segment()将 ELF 文件中所有的 Proghdr 中的内容拷贝到子进程的内存空间，并且将未填充部分刷 0. 关闭文件后，调用 copy_shared_pages()将共享页拷贝至子进程空间，随后使用系统调用 sys_env_set_trapframe()设置好子进程的寄存器状态。随后将子进程标记为 RUNNABLE，整个 spawn 过程就完成了。

Exercise 7

之前已经提到了，我们需要使用一个新的系统调用 sys_env_set_trapframe()来对子进程的寄存器状态进行设置，接下来我们需要先实现这个系统调用。在设置好 kern/syscall.c 的 syscall()的分发函数过后，我们需要实现 sys_env_set_trapframe()来完成这个系统调用。需要注意的是，这个函数首先需要保证设置完成后子进程仍然处于用户态，其次需要保证在子进程运行时中断

不能被阻塞。因此我们需要对设置的 Trapframe 进行以下修改：

```
e->env_tf = *tf;
e->env_tf.tf_eflags |= FL_IF;
e->env_tf.tf_ds = GD_UD | 3;
e->env_tf.tf_es = GD_UD | 3;
e->env_tf.tf_ss = GD_UD | 3;
e->env_tf.tf_cs = GD_UT | 3;
```

这样这个系统调用就完成了。目前已经可以通过 spawnhello 测试。

Exercise 8

目前我们在 fork()中处理内存拷贝基本上使用了 COW 技术。但是在文件系统中我们我们在每个用户进程内单独维护了一个 Fd 数组，并且每个 Fd 在文件系统管理进程与用户进程间共享。如果这时使用 COW 技术，那么如果一个子进程试图修改 Fd，则会发生 Fd 会被另行拷贝一份，这明显是错误的，而且可能造成文件系统不一致。在 spawn()中，我们根本没有考虑 Fd 数组的情况。

现在我们希望能够在 fork()和 spawn()中实现对 Fd 以及每个 Fd 对应的 data 页的共享，而且希望能够支持在其他需要共享库的情况。因此我们需要对这两个函数进行修改。具体修改的方式是使用 PTE 上的一个位 PTE_SHARE，如果一个页被标记了这个位，则需要在复制页时不实用 COW，而是直接拷贝 PTE。

修改 fork()的方法非常简单，只需要在 duppage()加一个判断即可，如果包含 PTE_SHARE，则直接以相同的权限位 map 到子进程。

修改 spawn()则略复杂。在 copy_shared_pages()函数中检查所有的页，如果包含 PTE_SHARE 则采用同样的方式。

处理权限啊位的时候需要注意应当使用 PTE&PTE_SYSCALL，而不是 PTE&0xFFF。因为后者会把 dirty 位等也包含进去。

```
for(addr = UTEXT; addr < (UXSTACKTOP - PGSIZE); addr += PGSIZE) {
    pde = &uvpd[PDX(addr)];
    pte = &uvpt[addr/PGSIZE];
    if(pde && *pde & PTE_P) {
        if(pte && (*pte & PTE_P) && (*pte & PTE_U)) {
            if(*pte & PTE_SHARE) {
                r = sys_page_map(
                    0, (void *)addr, child, (void *)addr, *pte & PTE_SYSCALL
                );
                if(r < 0)
                    return r;
            }
        }
    }
}
```

完成后已经可以通过 testpteshare 和 testfdsharing 测试

第三部分

在这一部分中，我们需要实现一个简单的 Shell。在此之前，我们需要能够支持在用户程序内可以使用键盘和串口进行输入。

Exercise 9

在之前我们能够对虚拟机进行输入输出的地方只有在 monitor 中。如果我们希望实现 Shell，就需要我们为普通进程添加输入输出的功能。在 lib/console.c 中，JOS 已经提供了一些函数，之前我们已经提到在文件系统中有一种 Dev 叫 devcons，这种文件类型就是控制台。在这个文件中定义了这种 Dev 的 read, write, open, close 等函数，以便用户进程使用文件读写的相同接口进行操作。这些函数其实是包装了一些读写的系统调用。

我们目前需要为 trap.c 添加处理中断 IRQ_OFFSET+IRQ_KBD 和 IRQ_OFFSET+IRQ_SERIAL，将它们分发到相应处理程序。这些程序就会将获得的输入存储到 lib/console.c 中 devcons 的 buffer 中。随后用户程序调用这个 Dev 的接口就可以读入内容了。

目前已经可以通过 testkbd 测试。

Exercise 10

我们可以看到 user/sh.c 的基本逻辑如下，首先初始化。随后进入一个无限循环，每次读入一行指令，随后使用 fork()创建子进程。在子进程内调用 runcmd()，随后退出。父进程继续循环。

函数 runcmd()的工作是根据指令执行有关操作。在这个 Exercise 中，我们需要实现输入的重定向，因此我们要修改的就是这个函数。

```
if ((fd = open(t, O_RDONLY)) < 0) {
    cprintf("open %s for write: %e", t, fd);
    exit();
}
if (fd != 0) {
    dup(fd, 0);
    close(fd);
}
break;
```

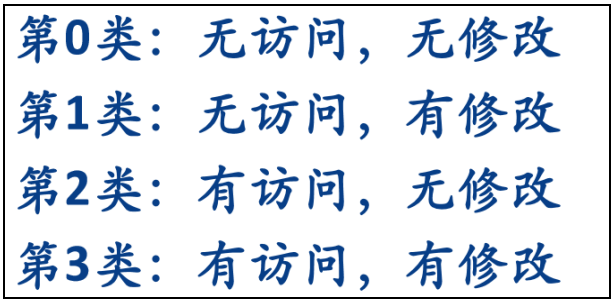
我们目前为止已经可以通过全部测试。

```
spawn via spawnhello: OK (1.4s)
PTE_SHARE [testpteshare]: OK (0.8s)
PTE_SHARE [testfdsharing]: OK (1.2s)
start the shell [icode]: OK (2.0s)
    (Old jos.out.icode failure log removed)
testshell: OK (3.1s)
    (Old jos.out.testshell failure log removed)
primespipe: OK (13.4s)
    (Old jos.out.primespipe failure log removed)
Score: 145/145
```

Challenge 2 - Cache Eviction Policy

如果内存页不足，就会导致文件系统的 Block Cache 成功映射新的磁盘块。我们可以在这种情况下发生时根据一定的页面置换策略从而将一些不太频繁使用的页置换回磁盘。

在操作系统课上，我们学习了许多页面置换算法。在这里我将要采用最近未使用算法（NRU）[i]，这个算法将内存页分为以下四种：



这四种置换的代价从低到高。因此置换的策略是从第 0 类开始，找到最小编号的页，随后置换，如果没有则再找第 1 类的页，以此类推。

我们可以从 PTE 的 dirty 位和 access 位来判断这一页是否有访问或者修改。如果在第一轮扫描中没有找到置换页，则在之后的扫描中，发现到一个页的 access 位被设置，就立刻把这一位改为 0.

由于我们不方便在用户程序内对页表进行修改，因此需要添加系统调用来清楚 access 位。我们把这个系统调用定为 SYS_page_clean_access。

```
int
compress()
{
    int i;
    for(i = 2; i < DISKSIZE/BLKSIZE; i++) {
        if(block_is_free(i))
            return i;
        if(!va_is_dirty(diskaddr(i)) && !va_is_accessed(diskaddr(i))) {
            free_block(i);
            return i;
        }
    }
    for(i = 2; i < DISKSIZE/BLKSIZE; i++) {
        if(block_is_free(i))
            return i;
        va_clean_accessed(diskaddr(i));
        if(!va_is_dirty(diskaddr(i))) {
            flush_block(diskaddr(i));
            free_block(i);
            return i;
        }
    }
    for(i = 2; i < DISKSIZE/BLKSIZE; i++) {
        if(block_is_free(i))
            return i;
        va_clean_accessed(diskaddr(i));
        if(!va_is_accessed(diskaddr(i))) {
            flush_block(diskaddr(i));
            free_block(i);
            return i;
        }
    }
    for(i = 2; i < DISKSIZE/BLKSIZE; i++) {
        if(block_is_free(i))
            return i;
        va_clean_accessed(diskaddr(i));
        return i;
    }


    return -E_NO_MEM;
}
```

内容三：遇到的困难以及解决方法

困难 1

这个 Lab 从以前的 Merge 和研究和写代码时都可以看出这个 Lab 经常有一些小问题，比较严重的我已经写在了意见和建议里。

内容四：收获及感想

终于完成了所有的五个 Lab，我个人感觉对于一些能力还是有很大的提高，比如阅读和理解复杂的代码，因为在 JOS 中很多系统非常复杂，想要通过阅读代码理解整个系统其实难度非常大。但是做 Lab 的前提是对整个系统的功能有准确而且全面的理解，否则就容易造成很多错误。我也在这方面吃了很多亏，经常由于理解不到位造成 Bug 找不出来。有的时候甚至已经到了后面的 Lab 又因为前面的 Bug 而出错。当然越到后面感觉越顺利，可能是因为已经对整个系统有了更为充分的理解。

内容五：对课程的意见和建议

文件系统的 IPC 过程中明显有 Bug，在 lib/file.c 中的 fsipc()函数在 ipc_send()函数后紧接着会 ipc_recv()，但是这时用户程序并不能分辨是谁在向自己发送信息。如果这个时候正好有其他进程发信息给自己，这个进程就无法收到文件系统管理程序发的信息了。

```
static int
fsipc(unsigned type, void *dstva)
{
    static envid_t fsenv;
    if (fsenv == 0)
        fsenv = ipc_find_env(ENV_TYPE_FS);

    static_assert(sizeof(fsipcbuf) == PGSIZE);

    if (debug)
        cprintf("[%08x] fsipc %d %08x\n", thisenv->env_id, type, *(uint32_t *)&fsipcbuf);

    ipc_send(fsenv, type, &fsipcbuf, PTE_P | PTE_W | PTE_U);
    return ipc_recv(NULL, dstva, NULL);
}
```

内容六：参考文献

[i] 操作系统课件 10-2016-春季-存储模型 3-发布版.pdf