JOS 实验五实验记录
作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室
（w f lfj and 1874）

开始本实验之前请使用 svn 把代码合并。

本实验编程技巧不多，不想 lab4，用了很多很多技巧，好像是在卖弄什么似的，但是代码量大和逻辑复杂，总的来说分成四大块，分别是服务器模块，用户模块还有底层模块和通讯模块。
以下先按照底层模块，服务器模块（包含通讯模块），用户模块（包含通讯模块），**四个模块如何合作**的顺序详细叙述。

底层模块（fs.c)
磁盘结构：

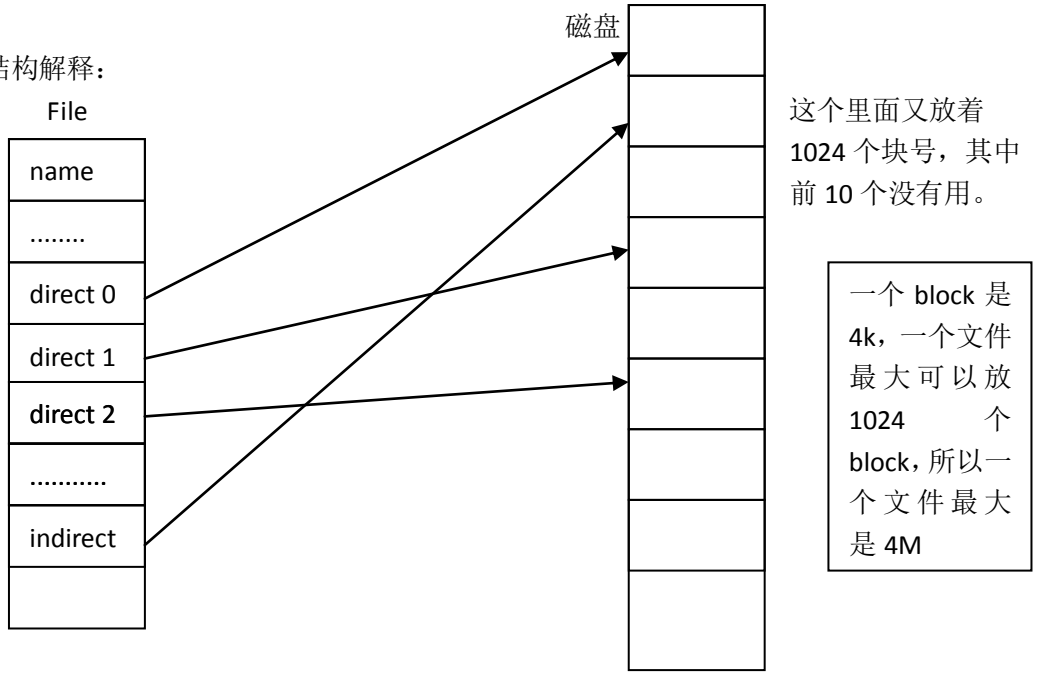
super 0	super 1	bitmap	Directory or file

super 0 这里我们不用管，是 bootloader。
super 1 记录着磁盘的基本信息（魔数和总块数）和根目录文件。
bitmap 记录这那些块可用那些不可用（已用或者未用）。

File 结构可以表示文件或者目录，在 jos 里面目录是记录文件和目录信息的文件。
如果一个文件是目录的话，那么这个文件里面记录的是 File 结构数组。

文件系统里面有两个概念，一个是实际块号，一个是文件块号，实际块号就是块在磁盘的位置，也可以在服务器进程中转化为虚拟地址。文件块号是相对文件而言的，文件的第一块块号为 0

File 结构解释：



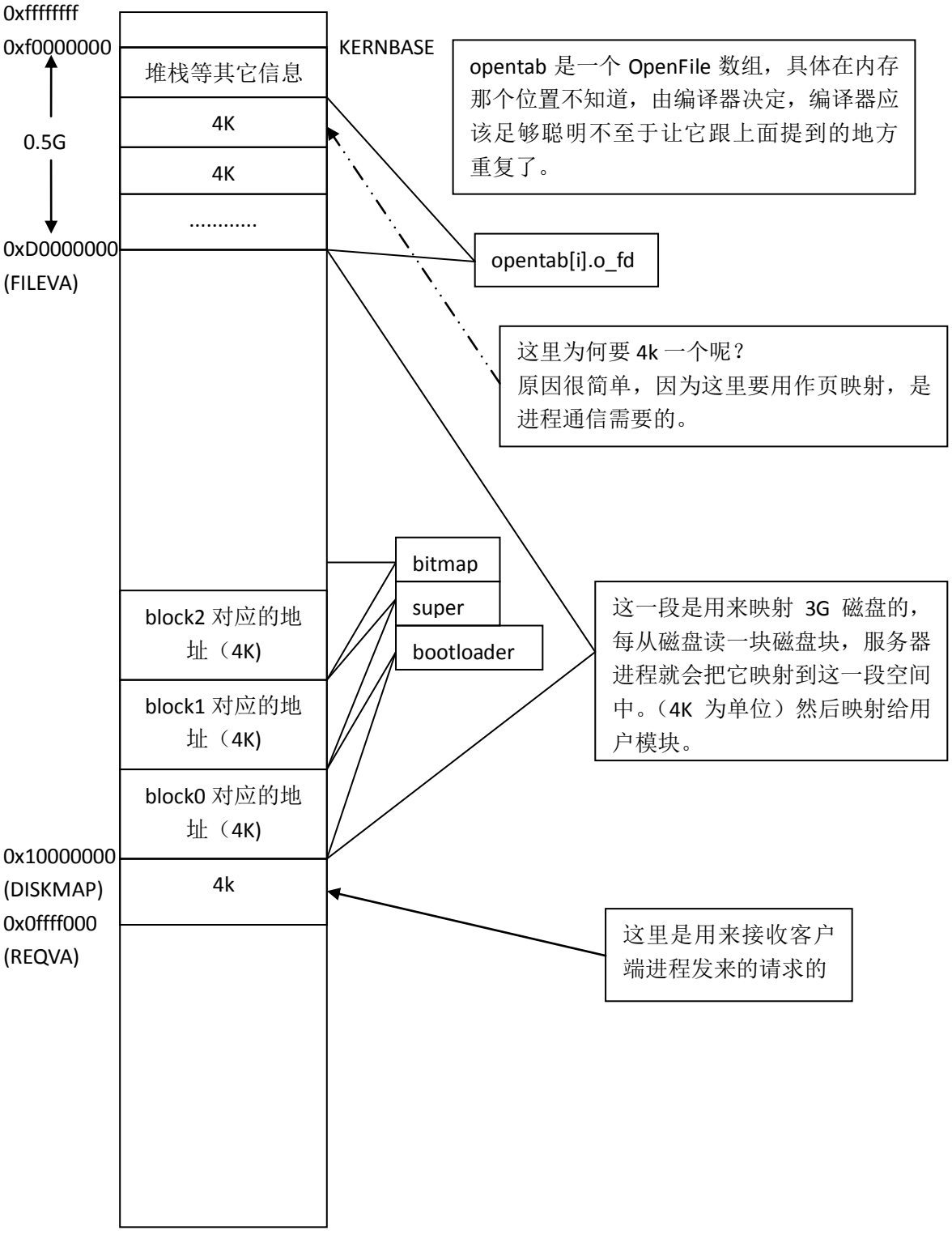
.....

。

如果文件的大小小于 10 个 block，那么文件块号就放在 direct 记录的块里面，如果文件大于 10 个 block，那么文件的 10 个块以上的文件块号就放在 indirect 里面，indirect block 是指向一个块的，块里面放在块号的信息，这里要注意的是 indirect block 块的前十个块号是没有用的，为什么作者要这样做，这是因为要保证文件最大是 4M，jos 文件最大只能是 4M。

服务器模块：

先看线性地址（虚拟地址）的分布情况



0x00000000
这里最关键的结构式 OpenFile 结构

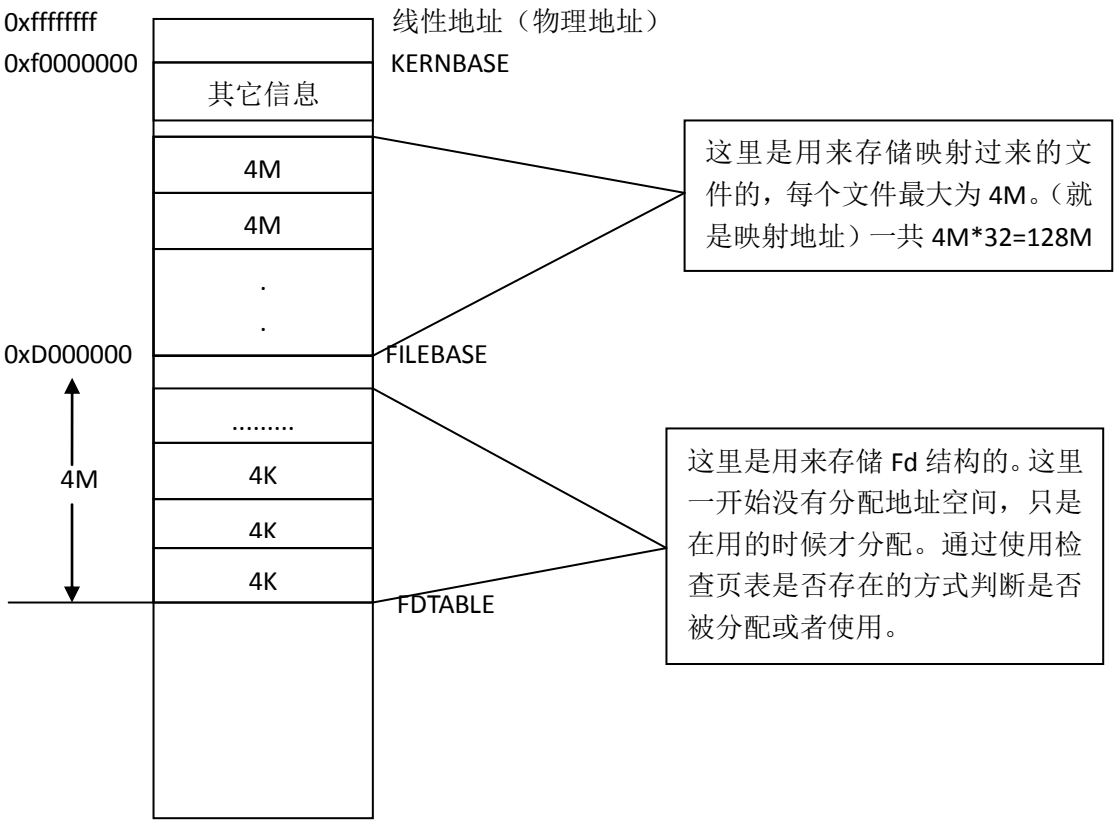
```
struct OpenFile {
        uint32_t o_fileid;
        struct File *o_file;
        int o_mode;
        struct Fd *o_fd;
};
```

其中 o_file 是用来操作底层文件系统的，
o_fd 是用来传到客户进程的，客户进程根据 o_fd 的信息操作文件。

```
void
serve_init(void)
{
        int i;
        uintptr_t va = FILEVA;
        for (i = 0; i < MAXOPEN; i++) {
                opentab[i].o_fileid = i;
                opentab[i].o_fd = (struct Fd*) va;
                va += PGSIZE;
        }
}
```

这段代码就是把 o_fd 映射到 FILEVA 以上的地址，以便进程之间的通讯，具体见服务器模块的图。

用户模块：



0x0000000
这里最关键的结构式 Fd
用户进程通过 Fd 保存的信息实现相关的文件操作。（例如 dev 是保存相关操作函数的指针）

通信模块：
在 inc/fs.h 中定义了一些通信的数据结构。

```
struct Fsreq_open {
        char req_path[MAXPATHLEN];
        int req_omode;
};

struct Fsreq_map {
        int req_fileid;
        off_t req_offset;
};
```

通过这些数据结构，服务器进程对底层文件系统进行相应的操作。
发送函数如下：

```
fsipc(FSREQ_OPEN, req, fd, &perm);
```

第一个参数是操作类型，第二个是要传递的参数，第三个就是 fd（服务器进程把 openfile 的 o_fd 映射到这里），第四个就是权限。

四个模块的合作

大体上是：用户需要文件操作--->用户进程向服务器进程发出请求--->服务器进程操作底层文件模块--->服务器进程向用户进程发出已经完成操作的信息，并把相关信息传递回去--->用户进程获得相关文件。

底层文件用 file 结构操作，用户用 fd 结构，服务器用 OpenFile 把两者联系起来。

先从客户端进程开始（icode）
客户端：
第一个函数（icode.c)

```
if ((fd = open("/motd", O_RDONLY)) < 0)
```

这个函数的作用就是打开一个文件并获得句柄。（file.c)

```
if ((r = fd_alloc(&fd)) < 0)
```

分配一个句柄(fsip.c)

```
int
fsipc_open(const char *path, int omode, struct Fd *fd)
{
        int perm;
        struct Fsreq_open *req;

        req = (struct Fsreq_open*)fsipcbuf;
        if (strlen(path) >= MAXPATHLEN)
                return -E_BAD_PATH;
        strcpy(req->req_path, path);
        req->req_omode = omode;

        return fsipc(FSREQ_OPEN, req, fd, &perm);
}
```

方框里面的内容是设置 req，serve 将根据 req 这个参数对底层文件系统进行操作。
然后发送到服务器端。这里 FSREQ_OPEN 是告诉服务器进程要进行什么样的操作，req 就是这个操作需要的参数，fd 是地址，服务器进程进行完相关的处理之后就会把内容映射到 fd 的地址中，fd 具体在哪里请看客户端模块的图。
现在进入 fsipc 函数，代码如下（fsipc.c)：

```
static int
fsipc(unsigned type, void *fsreq, void *dstva, int *perm)
{
        envid_t whom;

        if (debug)
                cprintf("[%08x] fsipc %d %08x\n", env->env_id,
 type, fsipcbuf);
        cprintf("type: %d\n",type);
        ipc_send(envs[1].env_id, type, fsreq, PTE_P | PTE_W |
PTE_U);
        return ipc_recv(&whom, dstva, perm);
}
```

具体操作是向服务器进程发送该发送的数据,注意，这里的 ipc_send 会循环发送，知道服务器进程响应为止。
然后进入阻塞状态，ipc_recv 函数。

现在进入服务器模块：
第一个函数：serve_init()　　(serv.c)

```
        serve_init();
        cprintf("serve_init success!!\n");
        fs_init();
        cprintf("fs_init success!!\n");
        fs_test();

        serve();
```

进入 serve_init 函数，此函数的功能是初始化服务器进程。

```
void
serve_init(void)
{
        int i;
        uintptr_t va = FILEVA;
        for (i = 0; i < MAXOPEN; i++) {
                opentab[i].o_fileid = i;
                opentab[i].o_fd = (struct Fd*) va;
                va += PGSIZE;
        }
}
```

初始化 opentab，并把 o_fd 映射到 FILEVA 以上的地址，具体详见服务器模块图。
然后进入底层文件系统初始化函数

底层模块：

```
fs_init();
```

这个函数的具体功能是读出 super 块很 bitmap

```
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);
        cprintf("read_super start!!\n");
        read_super();
        cprintf("read_super success!!\n");
        check_write_block();
        cprintf("check_write_block success!!\n");
        read_bitmap();
}
```

然后是 fs_test 函数，这个函数可以删除，这里就不讨论了。

退出底层模块：

现在进入 serve（）函数

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

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

                // 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...
                }

                switch (req) {
                case FSREQ_OPEN:
                        serve_open(whom, (struct Fsreq_open*)REQVA);
                        break;
                case FSREQ_MAP:
                        serve_map(whom, (struct Fsreq_map*)REQVA);
                        break;
                case FSREQ_SET_SIZE:
                        serve_set_size(whom, (struct Fsreq_set_size*)REQVA);
                        break;
                case FSREQ_CLOSE:
                        serve_close(whom, (struct Fsreq_close*)REQVA);
                        break;
                case FSREQ_DIRTY:
                        serve_dirty(whom, (struct Fsreq_dirty*)REQVA);
                        break;
                case FSREQ_REMOVE:
                        serve_remove(whom, (struct Fsreq_remove*)REQVA);
                        break;
                case FSREQ_SYNC:
                        serve_sync(whom);
                        break;
                default:
                        cprintf("Invalid request code %d from %08x\n", whom, req);
                        break;
                }
                sys_page_unmap(0, (void*) REQVA);
        }
}
```

这个函数是一个无限循环。

当运行这个函数的时候该进程会进入阻塞状态，然后等待用户模块向它发送信息，然后进行相关处理。（第一块代码是使进程进入阻塞状态，然后进入 switch，准备进行相应的处理）

按照上面用户模块发来的信息，应该是进入 serve_open：

```
case FSREQ_OPEN:
        serve_open(whom, (struct Fsreq_open*)REQVA);
```

现在进入 serve_open 函数（serv.c)

```
if ((r = openfile_alloc(&o)) < 0) {
```

先分配一个 openfile，openfile 到底是什么请看服务器端模块图。

现在进入**底层模块**：（fs.c)

```
if ((r = file_open(path, &f)) < 0) {
```

进入 file_open 函数

```
int
file_open(const char *path, struct File **pf)
{
        // Hint: Use walk_path.
        // LAB 5: Your code here.
        struct File *dir;
        int r;
        char lastelem[MAXNAMELEN];
        if( (r = walk_path(path, &dir, pf, lastelem)) < 0){
                return r;
        }
//      panic("file_open not implemented");
        return 0;
}
```

根据路径从磁盘中读出 file。并放在 pf 中。

退出**底层模块**。

进入**服务器模块**：

```
        o->o_file = f;

        // Fill out the Fd structure
        o->o_fd->fd_file.file = *f;
        o->o_fd->fd_file.id = o->o_fileid;
        o->o_fd->fd_omode = rq->req_omode;
        o->o_fd->fd_dev_id = devfile.dev_id;
        o->o_mode = rq->req_omode;

        if (debug)
                cprintf("sending success, page %08x\n", (uintptr_t) o->o_fd);
        ipc_send(envid, 0, o->o_fd, PTE_P|PTE_U|PTE_W|PTE_SHARE);
        return;
```

设置 Openfile，然后发送回用户模块。然后服务器模块返回 serve 函数，并进入阻塞状态，等待用户模块向它发送消息。

退出**服务器模块**

进入**用户模块**：

fsipc 函数（fsipc.c)

```
static int
fsipc(unsigned type, void *fsreq, void *dstva, int *perm)
{
        envid_t whom;

        if (debug)
                cprintf("[%08x] fsipc %d %08x\n", env->env_id, type, fsipcbuf);
        cprintf("type: %d\n",type);
        ipc_send(envs[1].env_id, type, fsreq, PTE_P | PTE_W | PTE_U);
        return ipc_recv(&whom, dstva, perm);
}
```

ipc_recv 函数返回，用户进程继续运行。

至此一次文件操作完成。

有些文件操作需要用到 dev 这个结构，这个结构就是把用户模块的文件基本操作函数指针放在里面。但是具体原理是一样的。

到此为止，文件系统的框架应该已经了解。下面进行深入理解。

开始做这个实验之前，我们必须先读相应的代码。
名词：
服务器进程指的是映射磁盘块的进程。

文件：fs.c
char* diskaddr(uint32_t blockno)
把块号转换成服务器进程内存相对应的虚拟地址。

bool va_is_mapped(void *va)
判断虚拟地址 va 在服务器进程中是否被映射。

bool block_is_mapped(uint32_t blockno)
判断 blockno 指定的 block 是否在服务器进程中被映射。

bool va_is_dirty(void *va)
判断 va 的内容是否被修改了

bool block_is_dirty(uint32_t blockno)
判断 blockno 对应的内容（这些内容必须应经映射到服务器进程）是否被修改。

int map_block(uint32_t blockno)
实际上就是分配一个页面个 blockno 对应的地址

static int read_block(uint32_t blockno, char **blk)
先映射一个页到服务器进程的地址空间，然后把 blockno 对应的块读到内存中。

void write_block(uint32_t blockno)
判断 blockno 对应的内存页是不是脏页，如果不是，直接返回，如果是，写到磁盘中。

void unmap_block(uint32_t blockno)
从服务器进程中把一个页的映射删除，但是这个页必须是干净的（相对于脏页）或者是 bitmap 里面标记为 free 的（被删除了）。

bool block_is_free(uint32_t blockno)
判断磁盘块对应的 bitmap 位是否为 0,1 表示已经用了，0 表示没有用。

int alloc_block_num(void)
分配一个磁盘块，返回磁盘块号，然后把 bitmap 写回磁盘。

int alloc_block(void)

分配一个磁盘块，然后把磁盘块映射到相应地址。

void read_super(void)
把超级块读到服务器进程的地址中。

void read_bitmap(void)
把 bitmap 读到服务器进程的地址空间中。

void check_write_block(void)
这个函数是用来检测 write_block 函数是否正常。通过把块 1 复制到块 0，然后修改块 1，写入磁盘，然后读出，然后比较，然后把块 0 的内容放回去，然后 super =块 1

void fs_init(void)
先检查有没有磁盘 1(还有一个是 0)，如果有设置 diskno 为 1，这个全局变量在 ide_read 里面会用到，用来判断写到那一个磁盘上。
读超级块。
检查写函数
读 bitmap

int file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
寻找文件 f 中块号为 filebno 的块（文件中第一块为 0），ppdiskbno 是指向 f_direct 数组的元素或者 f_indirect 指向的块的元素。
alloc 是没有块的时候是否分配一个新块。
*ppdiskbno 是实际块号的地址
这里有一句代码：

```
f = &super->s_root;
```

因为 super 记录着文件系统的根目录的文件的。

int file_map_block(struct File *f, uint32_t filebno, uint32_t *diskbno, bool alloc)
给文件 f 的第 filebno 个块映射。
alloc 是否需要新建一个块
diskbno 返回的实际块号。

int file_clear_block(struct File *f, uint32_t filebno)
清除 f 的一个块。

int file_get_block(struct File *f, uint32_t filebno, char **blk)
从磁盘读一个块，并且映射到对应的服务器进程空间。
*blk 为虚拟内存地址。
如果没有就分配一个块。

int file_dirty(struct File *f, off_t offset)
使 offset/BLKSIZE 个块为脏读，通过自己写自己，然后 cpu 自动在内存中标记为脏。

int dir_lookup(struct File *dir, const char *name, struct File **file)
在 dir 中找到为 name 的文件，并把它放到*file 中

int dir_alloc_file(struct File *dir, struct File **file)
在指定的目录 dir 下分配一个 file

static inline const char* skip_slash(const char *p)
略过 /

static int walk_path(const char *path, struct File **pdir, struct File **pf, char *lastelem)
根据路径遍历文件系统，如果有文件*pf = 要找的文件
如果没有，把剩下的路径复制到 lastelem 中。

int file_create(const char *path, struct File **pf)
建立一个 path 文件

int file_open(const char *path, struct File **pf)
打开一个文件。

static void file_truncate_blocks(struct File *f, off_t newsize)
把文件大小变为 newsize

int file_set_size(struct File *f, off_t newsize)
把文件 f 设置成 newsize 大小

void file_flush(struct File *f)
把文件的内容同步到磁盘上。这里可以通过 file_map_block 查找实际块号。

void fs_sync(void)
同步硬盘上所有数据。

void file_close(struct File *f)
关闭一个文件

int file_remove(const char *path)
删除一个文件

文件系统相对前面的虚拟内存系统来说是相对简单的，下面开始实验：
EX1:
在 env.c 的 env_alloc 中添加以下代码：

```
    e->env_tf.tf_eflags |= FL_IF;
    if( e == &envs[1]){
            e->env_tf.tf_eflags |= FL_IOPL_3;
    } else {
            e->env_tf.tf_eflags |= FL_IOPL_0;
    }
    // Clear the page fault handler until user
```

这里涉及一些 cpu 的硬件特性，eflag 的 FL_IOPL 位，FL_IOPL 表明当前段（任务）的特权级为多少才可以访问 IO，如果是 3，就表明当前段（任务）的特权级为 3,2,1 都可以访问 I/O，如果为 0，就表明当前段的特权级必须为 0 才能访问 I/O。

EX2:

read_block 代码如下：

```
        // LAB 5: Your code here.
        addr = diskaddr(blockno);
        if( blk != NULL){
                *blk = addr;
        }
        // here is to check that is the va is already in the virtual memory.
//      cprintf("va_is_mapped start!!\n");
//      cprintf("addrress: 0x%x\n",addr);
        if( va_is_mapped( addr) == 1){
//              cprintf("va is mapped!!\n");
                return 0;
        }
//      cprintf("va isn't map!!\n");
        if((r = map_block(blockno)) < 0){
                return r;
        }
//      cprintf("map_block success!!\n");
        if( (r = ide_read(blockno * BLKSECTS, addr, BLKSECTS))< 0){
                return r;
        }
//      cprintf("ide_read success!!\n");
//      panic("read_block not implemented");
        return 0;
}
```

如果 addr 已经在内存中有映射，函数返回，不用任何操作。如果没有，根据 blockno 分配内存页，并进行影射，然后把磁盘上的数据映射到内存上。blockno 对应那一个内存地址，根据服务器模块图理解。

根据上面的函数解释看这里的代码应该很容易懂的，这里不再累述。

write_block

```
        // Write the disk block and clear PTE_D.
        // LAB 5: Your code here.
        addr = diskaddr(blockno);
        if( va_is_dirty(addr) == 0){
                cprintf("va isn't dirty!!\n");
                return;
        }
        if(ide_write(blockno * BLKSECTS, addr, BLKSECTS) < 0){
                panic("write_block ide_write fail!!\n");
        }
        //set the pte_d == 0
        if(sys_page_map(0, addr, 0, addr, PTE_USER) < 0){
                panic("write_block sys_page_map fail!!\n");
        }
        return ;
```

先判断是否是脏页，如果是，就写入，并清除 PTE_D 位，如果不是直接返回。

这里 PTE_D 位是什么时候设置的呢？这又是 cpu 做的事情，当一个内存也被访问的时候，cpu 就会对其进行标记，把 PTE_D 位置 1，但是 cpu 从来不会主动把 PTE_D 位置 0，所以必须由程序设定。

这里可以用 sys_page_map 来清除 PTE_D，PTE_D 是脏读位。

EX3:

read_bitmap

```
        // LAB 5: Your code here.
//      panic("read_bitmap not implemented");
        bitmap_blkno = super->s_nblocks / BLKBITSIZE;
        if( super -> s_nblocks % BLKBITSIZE == 0){
        } else {
                bitmap_blkno ++;
        }
        for( i = 0; i < bitmap_blkno; i++){
                if(read_block(2 + i, &blk) < 0){
                        panic("read_bitmap: read_block fail!!\n");
                }
        }
        bitmap = (uint32_t *)diskaddr(2);
        // Make sure the reserved and root blocks are marked in-use.
        assert(!block_is_free(0));
        assert(!block_is_free(1));
        assert(bitmap);

        // Make sure that the bitmap blocks are marked in-use.
        // LAB 5: Your code here.
        for( i = 0; i < bitmap_blkno; i++){
                assert(!block_is_free(2 + i));
        }
        cprintf("read_bitmap is good\n");
}
```

第一块代码是用来求出一共需要多少块才能存储 bitmap。

第二块代码是用来检查存储 bitmap 的块是否标记为已用，在 bitmap 里面 0 表示没有用，1 表示已经用了。

EX4:

alloc_block_num

```
        // LAB 5: Your code here.
/       panic("alloc_block_num not implemented");i
        int i;
        for( i = 3; i< super->s_nblocks; i++){
                if( (bitmap[i / 32] & (1 << ( i % 32))) != 0){
                        bitmap[i /32] &= ~(1 << ( i % 32));
                        write_block(2 + i/BLKBITSIZE);
                        return i;
                }
        }
        return -E_NO_DISK;
```

这个函数是从 bitmap 中找一个空闲的块，然后分配出去，但是这里要注意，一旦 bitmap 被修改了，必须马上写入磁盘，保证同步。

这里 bitmap 的每一位表示一个块，所以要用到位操作，具体为何这样写，自己模仿程序走一下就会知道。

EX5:

file_open

```
    // LAB 5: Your code here.
    struct File *dir;
    int r;
    char lastelem[MAXNAMELEN];
    if( (r = walk_path(path, &dir, pf, lastelem)) < 0){
            return r;
    }
    panic("file_open not implemented");
    return 0;
```

打开一个文件。

file_get_block

```
    // Hint: use file_map_block and read_block.
    // LAB 5: Your code here.
    if( ( r = file_map_block(f, filebno, &diskbno, 1)) < 0){
            return r;
    }
    if( ( r = read_block(diskbno, blk)) < 0){
            return r;
    }
    panic("file_get_block not implemented");
    return 0;
```

取得一个 block

file_truncate_blocks：

```
    // LAB 5: Your code here.
    old_nblocks = ROUNDUP(f ->f_size, BLKSIZE) / BLKSIZE;
    new_nblocks = ROUNDUP(newsize, BLKSIZE) / BLKSIZE;
    for( bno = new_nblocks; bno < old_nblocks; bno ++){
            file_clear_block(f,bno);
    }
    if(f ->f_indirect != 0 && new_nblocks <= NDIRECT){
            free_block(f->f_indirect);
            f->f_indirect = 0;
    }
    panic("file_truncate_blocks not implemented");
```

把文件的大小设置为新大小。

file_flush

```
void
file_flush(struct File *f)
{
        // LAB 5: Your code here.
        int blkno = ROUNDUP(f -> f_size, BLKSIZE) / BLKSIZE;
        int i,disk_blk_no;
        for( i = 0; i < blkno; i++){
                if(file_map_block(f, i,(unsigned int *)(&disk_blk_no), 0) < 0){
                        panic("file_flush:file_map_block fail!!\n");
                }
                if(block_is_dirty(disk_blk_no) != 0){
                        write_block(disk_blk_no);
                }
        }
//      panic("file_flush not implemented");
}
```

把文件同步到磁盘上，就是写入磁盘。

以上为底层文件系统的实现。

----------好累啊!! ----------^_^

现在进入服务器进程

serv.c

void serve_init(void)

这个函数初始化服务器，实际上就是初始化 opentab 和完成 o_fd 的映射。

int openfile_alloc（struct OpenFile **o）

```
// Allocate an open file.
int
openfile_alloc(struct OpenFile **o)
{
        int i, r;

        // Find an available open-file table entry
        for (i = 0; i < MAXOPEN; i++) {
                switch (pageref(opentab[i].o_fd)) {
                case 0:
                        if ((r = sys_page_alloc(0, opentab[i].o_fd, PTE_P|PTE_U|PTE_W)) < 0)
                                return r;
                        /* fall through */
                case 1:
                        opentab[i].o_fileid += MAXOPEN;
                        *o = &opentab[i];
                        memset(opentab[i].o_fd, 0, PGSIZE);
                        return (*o)->o_fileid;
                }
        }
        return -E_MAX_OPEN;
}
```

这个函数的无论什么情况都会执行 case 1 的内容，但是 case 0 的内容不一定会执行。
当什么时候会执行 case 0 呢，就是没有分配 o_fd 页的时候，o_fd 对应的地址在服务器模块的图中可以看到。
但是 case 1 又是怎么解释。
当分配一个页到 o_fd 的时候，pp_ref = 1，当它映射到用户进程的时候，pp_ref = 2，当用户进程关闭或者关闭文件的时候，pp_ref = 1，因为用户模式中的 fd_close 中是会 unmap 页的。

```
(void) sys_page_unmap(0, fd);
```

但是在服务器进程中 serve_close 是不会把 page unmap 的，也就是 pp_ref 是不会等于 0 的。所以当再次调用这个函数的时候，就不用重新分配页。这样可以提高速度，但是如果在前一分钟打开 30 个文件，在之后的时间只打开一个文件的话，会造成页的浪费。如果要解决这个问题的话可以在 serve_close 中加上 page unmap。

int openfile_lookup(envid_t envid, uint32_t fileid, struct OpenFile **po)
这个函数就是根据 fileid 寻找 openfile。

void serve_open(envid_t envid, struct Fsreq_open *rq)
打开一个文件，并填充 OpenFile 中的信息，然后发回去。

void serve_set_size(envid_t envid, struct Fsreq_set_size *rq)
设置文件的大小

void serve_map(envid_t envid, struct Fsreq_map *rq)
把服务器进程的一个 block 映射到用户进程。

void serve_close(envid_t envid, struct Fsreq_close *rq)
关闭一个文件。

void serve_remove(envid_t envid, struct Fsreq_remove *rq)
删除一个文件。

void serve_dirty(envid_t envid, struct Fsreq_dirty *rq)
把一个页标记为脏页

void serve_sync(envid_t envid)
把所有文件同步到磁盘。

void serve(void)
服务器进程主要函数，用来分配用户进程发来的请求。

EX6 （serv.c)
void serve_map(envid_t envid, struct Fsreq_map *rq)

```
        // LAB 5: Your code here.
        if( (r = openfile_lookup(envid, rq->req_fileid, &o)) < 0){
                ipc_send(envid, r, 0, 0);
                return ;
        }
        if((r = file_get_block(o->o_file, ROUNDUP(rq->req_offset, BLKSIZE)/ BLKSIZE , &blk)) < 0){
                ipc_send(envid, r, 0, 0);
                return;
        }
        if(( o -> o_mode & O_ACCMODE) == O_RDONLY)
                perm = PTE_P | PTE_U;
        else
                perm =  PTE_P | PTE_U | PTE_W;
        ipc_send(envid, 0, blk, perm);
        return;

//      panic("serve_map not implemented");
```

这个函数把一个 block 映射到用户进程。

void serve_close(envid_t envid, struct Fsreq_close *rq)

```
        // LAB 5: Your code here.
//      panic("serve_close not implemented");
        if( ( r = openfile_lookup( envid, rq->req_fileid, &o)) < 0){
                ipc_send(envid, r, 0, 0);
                return;
        }
        cprintf("serve_close!!!!!!!!!!!!\n");
        file_close(o->o_file);
        ipc_send(envid, 0, 0, 0);
        return;
}
```

这个函数关闭一个文件。这里我们应该把所有页映射删除，但是为了提高效率，我们可以不做这一步。但是这样会造成物理页不够用。这里可以在 fs.c 中加入一个 file_unmap，然后把关闭的文件 unmap 掉，这样会好一点，但是考虑到 jos 就算在这里加入减少物理内存页的使用，也是无法解决它物理内存页分配完就会挂掉的问题，而且作者也没有要求，所以这里就不加了。

void serve_remove(envid_t envid, struct Fsreq_remove *rq)

```
// LAB 5: Your code here.
panic("serve_remove not implemented");
memmove(path, rq->req_path, MAXPATHLEN);
path[MAXPATHLEN - 1] = 0;
r = file_remove(path);
ipc_send(envid, r, 0, 0);
return;
```

这个函数的功能是删除一个文件。

void serve_dirty(envid_t envid, struct Fsreq_dirty *rq)

```
        // LAB 5: Your code here.
//      panic("serve_dirty not implemented");
        if(( r = openfile_lookup(envid, rq->req_fileid, &o)) <0){
                ipc_send(envid, r, 0, 0);
                return;
        }
        if(( r = file_dirty(o->o_file, rq->req_offset)) < 0){
                ipc_send(envid, r, 0, 0);
                return;
        }
        return;
}
```

把一个页标记为脏页。

现在进入用户模块：（fd.c)
INDEX2DATA(i)这个宏是把 fd 装换成放数据的地方。具体情况用户模块图。

char* fd2data(struct Fd *fd)
得到对应 fd 的映射的数据的地址，具体请看用户模块图。

int fd2num(struct Fd *fd)
fd 转成索引号。

int fd_alloc(struct Fd **fd_store)
分配一个 fd（这里可以翻译成文件句柄吧）。

int fd_lookup(int fdnum, struct Fd **fd_store)
寻找一个 fd。

int fd_close(struct Fd *fd, bool must_exist)
关闭一个文件句柄

int dev_lookup(int dev_id, struct Dev **dev)
根据 dev_id 寻找对应的 dev，这里只有一个 dev，dev 里面放的是磁盘基本操作的函数的地

址。

int close(int fdnum)
关闭一个句柄，注意这里有 page_unmap。

void close_all(void)
关闭所有文件。

int dup(int oldfdnum, int newfdnum)
复制一个文件句柄，并把对应的内容也映射到新的文件句柄的相应的数据区(FD2DATA）。具体请看用户模块图。

ssize_t read(int fdnum, void *buf, size_t n)
从文件中读数据到 buf。

ssize_t readn(int fdnum, void *buf, size_t n)
这个函数暂时没有用。

ssize_t write(int fdnum, const void *buf, size_t n)
写入文件。

int seek(int fdnum, off_t offset)
设置 fd 的 offset

int ftruncate(int fdnum, off_t newsize)
修改文件的大小。

int fstat(int fdnum, struct Stat *stat)
设置 stat 结构。

int stat(const char *path, struct Stat *stat)
设置 stat 结构。

EX7　fd.c
int fd_alloc(struct Fd **fd_store)

```
        // LAB 5: Your code here.

        //panic("fd_alloc not implemented");
        //return -E_MAX_OPEN;
        int i;
        for (i = 0; i < MAXFD; i++) {
                *fd_store = INDEX2FD(i);
//              cprintf("i:%d\n",i);
                if((vpd[PDX(*fd_store)] & PTE_P) == 0 ||
                        (vpt[VPN(*fd_store)] & PTE_P) == 0){
                        cprintf("fdnum:%d\n",i);
                        return 0;
                }
        }
        *fd_store = 0;
        return -E_MAX_OPEN;
}
```

通过检查页是否分配来确定 fd 有没有被分配。

int fd_lookup(int fdnum, struct Fd **fd_store)

```
{
        // LAB 5: Your code here.
        if( fdnum < 0 || fdnum > MAXFD){
        //      panic("fd_lookup fail!!\n");
                return -E_INVAL;
        } else {
//              cprintf("1234567890\n");
                *fd_store = INDEX2FD(fdnum);
        }
        if((vpd[PDX(*fd_store)] & PTE_P) == 0 || (vpt[VPN(*fd_store)] & PTE_P) == 0){
//              cprintf("********---------******\n");
                return -E_INVAL;
        }else {
//              cprintf("<<<<<<<---------->>>>>>\n");
                return 0;
        }
//      panic("fd_lookup not implemented");
        return -E_INVAL;
}
```

根据 fdnum 查找一个 fd。

EX 8

file.c

open 函数

```
//      panic("open() unimplemented!");
        struct Fd * fd;
        int r;
        if ((r = fd_alloc(&fd)) < 0)
                return r;
        cprintf("path: %s\n", path);
        if ((r = fsipc_open(path, mode, fd)) < 0)
                return r;
        if ((r = fmap(fd, 0, fd->fd_file.file.f_size)) < 0) {
                fd_close(fd, 1);
                return r;
        }
        cprintf("fd->fd_file.file.f_size:%d\n",fd->fd_file.file.f_size);
        return fd2num(fd);
}
```

这里要注意啦，open 的时候会把文件的内容都 map 到 fd 对应的数据区里面，具体请看用户

模块图，文件最大为 4M。

下面是 dev 结构，记录着基本操作函数的指针。

```
struct Dev devfile =
{
        .dev_id =       'f',
        .dev_name =     "file",
        .dev_read =     file_read,
        .dev_write =    file_write,
        .dev_close =    file_close,
        .dev_stat =     file_stat,
        .dev_trunc =    file_trunc
};
```

EX 9　file.c

static int file_close(struct Fd *fd)

```
        // LAB 5: Your code here.
//      panic("close() unimplemented!");
        int r;
        if ((r = funmap(fd, fd->fd_file.file.f_size, 0, 1)) < 0)
                return r;
        if ((r = fsipc_close(fd->fd_file.id)) < 0)
                return r;
        return 0;
```

这里会把 fdunmap 掉，跟 serv.c 是不一样的。然后 fd 对应数据的映射也会被 unmap（）。
funmap 会使相应文件的所有页变为脏，并且 unmap 掉。
然后 fsipc_close 会令服务器把所有脏页写回磁盘。

EX10

spawn 函数

这个函数的作用是把程序从磁盘读出，并放到 jos 上运行。
加载的时候请注意：可写的段需要分配物理页，不可写的直接映射就可以了。
具体代码如下：

```
    int r;
    int fdnum;
    if((fdnum = open(prog, O_RDWR)) < 0){
            return fdnum;
    }
    cprintf("fdnum : %d\n",fdnum);
    read(fdnum, elf_buf, 512);
    struct Elf * elfhdr = (struct Elf *) elf_buf;
    cprintf("elfhdr -> e_magic :0x%x\n", elfhdr ->e_magic);
    if(elfhdr -> e_magic != ELF_MAGIC){
            panic("magic num error!!\n");
    }
    if((child = sys_exofork()) < 0)
            return child;
    child_tf = envs[ENVX(child)].env_tf;
    child_tf.tf_eip = elfhdr -> e_entry;
    if(( r = init_stack(child, argv, &child_tf.tf_esp)) < 0){
            return r;
    }
    cprintf("breakpoint 1\n");
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr *)((uint32_t)elfhdr + elfhdr->e_phoff);
    eph = ph + elfhdr->e_phnum;
    for(;ph < eph; ph++){
            if(ph ->p_type != ELF_PROG_LOAD)
                    continue;
            if (ph->p_filesz > ph->p_memsz)
                    panic("spawn: invalid program header\n");
            if ((ph->p_flags & ELF_PROG_FLAG_WRITE) == 0) {
                    uint32_t start = ROUNDDOWN(ph->p_offset, PGSIZE);
                    uint32_t end = ROUNDUP(ph->p_filesz + ph->p_offset, PGSIZE);
                    uint32_t va = ROUNDDOWN(ph->p_va, PGSIZE);
                    uint32_t i;
                    void * blk;
                    // read every page of ph and then map it into child
                    for (i = start; i < end; i += PGSIZE) {
                            if ((r = read_map(fdnum, i, &blk)) < 0)
                                    return r;
```

```
                    if ((r = sys_page_map(0, blk, child, (void *)(va + (i - start)), PTE_P | PTE_U)) < 0)
                            return r;
            }
    }else {
            uint32_t start = ROUNDDOWN(ph->p_offset, PGSIZE);
            uint32_t end = ROUNDUP(ph->p_memsz + ph->p_offset, PGSIZE);
            uint32_t va = ROUNDDOWN(ph->p_va, PGSIZE);
            uint32_t i;
            void * blk;
            // seek is important, because using read we need to locate the file read/write pointer right
            seek(fdnum, ph->p_offset);
            // read every page
            for (i = start; i < end; i += PGSIZE) {
                    if ((r = sys_page_alloc(0, UTEMP, PTE_P | PTE_U | PTE_W)) < 0)
                            return r;
                    memset(UTEMP, 0, PGSIZE);
                    // see if is non-loaded portions
                    if (i < ph->p_offset + ph->p_filesz) {
                            // see if the page being read contains non-loaded portions
            //              if (ph->p_offset + ph->p_filesz - i >= PGSIZE)
                                    r = read(fdnum, UTEMP, PGSIZE);
            //              else
                                    r = read(fdnum, UTEMP, ph->p_offset + ph->p_filesz - i);
                            if (r < 0)
                                    return r;
                    }
                    if ((r = sys_page_map(0, UTEMP, child, (void *)(va + (i - start)), PTE_P | PTE_U | PTE_W)) < 0)
                            return r;
                    if ((r = sys_page_unmap(0, UTEMP)) < 0)
                            return r;
            }
    }

}
cprintf("breakpoint2\n");

if ((r = sys_env_set_trapframe(child, &child_tf)) < 0){
```

```
                cprintf("breakpoint3\n");
                return r;
        }
        if ((r = sys_env_set_status(child, ENV_RUNNABLE)) < 0){
                cprintf("breakpoint4\n");
                return r;
        }
        cprintf("breakpoint5\n");
        return child;
//      panic("spawn unimplemented!");

}
```

EX11

spawn.c

init_stack 函数

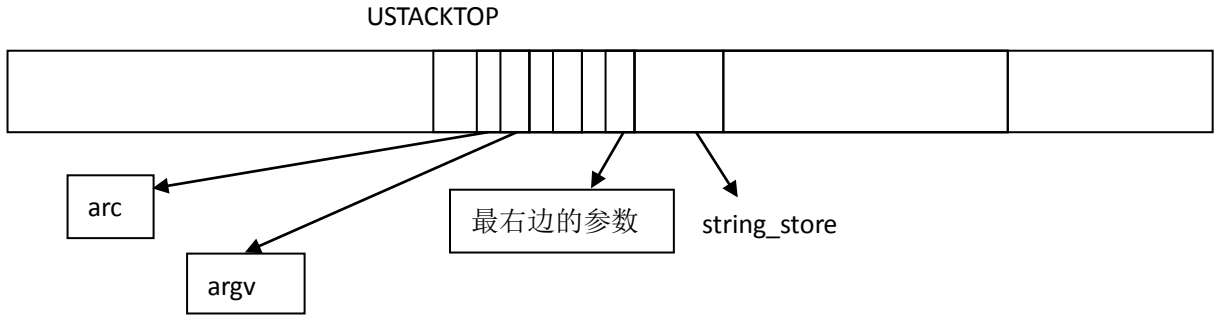
```
        // LAB 5: Your code here.
//      *init_esp = USTACKTOP;  // Change this!
        for (i = 0; i < argc; i++) {
                strcpy(string_store, argv[i]);
                argv_store[i] = UTEMP2USTACK(string_store);
                string_store += strlen(argv[i]) + 1;
        }
        argv_store[argc] = 0;
        *(argv_store - 1) = UTEMP2USTACK(argv_store);
        *(argv_store - 2) = argc;
        *init_esp = UTEMP2USTACK(argv_store - 2);
        // After completing the stack, map it into the child's address space
        // and unmap it from ours!
        if ((r = sys_page_map(0, UTEMP, child, (void*) (USTACKTOP - PGSIZE), PTE_P | PTE_U | PTE_W)) < 0)
                goto error;
        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                goto error;

        return 0;

error:
        sys_page_unmap(0, UTEMP);
        return r;
```

这个函数就是初始化我们运行的程序的堆栈。至于为什么要这样初始化，请了解编译器的内容，参数压栈是从右到左的。

初始化后的堆栈如下：



完成上述代码以后还要加几个系统中断。

这里为何要这么做请了解前面的实验。

```
  case SYS_env_set_trapframe:
          return sys_env_set_trapframe((envid_t)a1,(struct Trapframe *)a2);
  case SYS_env_set_status:
          return sys_env_set_status((envid_t)a1,(int)a2);
```

```
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
        // LAB 4: Your code here.
        // Remember to check whether the user has supplied us with a good
        // address!
        struct Env * env;
        int r;
        if ((r = envid2env(envid, &env, 1)) < 0)
                return r;
        env->env_tf = *tf;
        env->env_tf.tf_eflags |= FL_IF;
        env->env_tf.tf_cs |= 3;
        env->env_tf.tf_ds |= 3;
        env->env_tf.tf_es |= 3;
        env->env_tf.tf_ss |= 3;
        return 0;
//      panic("sys_set_trapframe not implemented");
}
```

```
static int
sys_env_set_status(envid_t envid, int status)
{
        // Hint: Use the 'envid2env' function from kern/env.c to translate an
        // envid to a struct Env.
        // You should set envid2env's third argument to 1, which will
        // check whether the current environment has permission to set
        // envid's status.

        // LAB 4: Your code here.
        struct Env *env;
        if( status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE){
                cprintf("parameter error!!\n");
                return -E_INVAL;
        }
        if( envid2env(envid, &env, 1) < 0){
                return -E_BAD_ENV;
        }
        env -> env_status = status;
        return 0;
//      panic("sys_env_set_status not implemented");
}
```

然后再 init.c 加上以下代码：

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

        // Start init
#if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE2(TEST, TESTSIZE);
        cprintf("test!!\n");
#else
        // Touch all you want.
//      ENV_CREATE(user_writemotd);
        // ENV_CREATE(user_testfsipc);
        ENV_CREATE(user_icode);
#endif
```

实验完成。