# Lab5 实验梳理

## 一、读写设备内容

首先实现设备的读写函数，由内存映射技术，对设备的访问其实就是对固定物理地址空间的访问。

### 1.1 sys_write_dev()

**函数用法：**

```
int sys_write_dev(u_int va, u_int pa, u_int len)
```

**参数：**

- va: 待写入数据的虚拟地址
- pa:带写入设备对应物理地址
- len:代写入设备对应长度

**返回值：**

成功返回0，否则返回错误码

**逻辑：**

首先进行长度、虚拟地址与物理地址的合法性判断；再使用memcpy函数，将一个代写入数据复制到设备对应的物理地址处。

**注意：**

```
+ len 只能为1、2或4
+ 对于外设访问中的物理地址，直接访问需要加KSEG1
```

### 1.2 sys_read_dev()

**函数用法：**

```
int sys_read_dev(u_int va, u_int pa, u_int len)
```

参数，返回值，逻辑与注意事项与上一函数相同。

唯一的不同是memcpy中调换 `va` 与 `pa + KSEG1` 的位置，由写变读。

## 1.3 ide_read()

**函数用法：**

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
```

**参数：**

- diskno:待读磁盘编号
- secno:待读取扇区的起始块编号
- dst:保存读入数据的内存起始位置
- nsecs:读取扇区数量

**逻辑：**

按位设置扇区的操作扇区数、扇区号、磁盘号、寻址模式等信息；之后循环读取所有扇区、每个扇区对应的读取过程中，循环调用 `syscall_read_dev()` 读取每个字节。

**注意：**

该函数中两处调用 `wait_ide_ready()` ，分别为写入操作信息之前，以及写入操作信息之后、读取内容之前。

可以认为上述的读和写分别为一个完整操作，每个操作执行前都需等待外设数据操作的完成；而写操作中其实多次调用 `syscall_write_dev()` 但它们之间并没有等待，因为这些操作之间互不影响。即使进行下一次操作时上一次操作还未完成，也不会引发什么错误。

> 🌈
>
> 在磁盘读写的流程中，我们需要反复检查 IDE 设备是否已经就绪。**这是由于 IDE 外设一般不能立即完成数据操作**，需要 CPU 检查 IDE 状态并等待操作完成。

## 1.4 ide_write()

**函数用法：**

```
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
```

参数、逻辑、注意等与上一函数完全一致。

唯一不同的大概就是work mode了吧(read / write)

# 二、文件系统结构

## 2.1 free_block()

**函数用法：**

```
void free_block(u_int blockno)
```

**参数：**

- blockno:待操作磁盘块编号

**逻辑：**

首先检查磁盘号是否正确（非0且在超级块记录的磁盘块数量范围内），然后将位图中对应位置1.（1表示空闲）

**注意：**

位图数组 `bitmap[]` ,一位代表一个磁盘块，所以 `blockno` 对应的位为 `bitmap` 数组的第 `blockno / 32` 个元素的第 `blockno % 32` 位。

## 2.2 create_file()

**函数用法：**

```
struct File *create_file(struct File *dirf)
```

**参数：**

- dirf:新建文件所在的目录（由于文件系统中的目录也是文件，所以类型还是struct File）

**返回值：**

- 新建文件的文件控制块

**逻辑：**

最终目的是在给定的目录下新建一个文件。实质上就是返回一个在该目录下的文件控制块，而且此时这个文件控制块应该是新的。

寻找该文件控制块的方法如下：首先遍历文件夹下已有的文件控制块，检查是否有已被回收的，如果有就重复利用之。如果没有就新增一个文件控制块。这两种情况的寻找逻辑分别是：

- `dirf->f_size / BLOCK_SIZE`获取文件控制块数并遍历，前十个在直接指针中寻找，后面的读取磁盘内容；
    - 然后遍历这个磁盘块中的8个文件控制块，找到空的（f_name[0]=='\0'）
- 如果上述方法没找到，说明需要一个新的文件块。通过`make_link_block()`函数申请新的磁盘块

## 2.3 make_link_block()（简述）

**函数用法：**

```
int make_link_block(struct File *dirf, int nblk)
```

为一个目录新增一个磁盘块，输入目录的文件控制块与已有的磁盘块数，返回新磁盘块的disk数组索引`bno`。

函数中增加`dirf`的`f_size()`，并添加`dirf`与`bno`的联系。（调用`save_block_link()`其实就是对直接/间接指针的赋值，使得后续能通过该指针正确找到此磁盘块）

## 2.4 disk_addr()

**函数用法：**

```
void *disk_addr(u_int blockno)
```

**参数：**

- blockno：待寻址的磁盘号

**逻辑：**

在文件服务进程的地址空间中，磁盘被映射到固定的虚拟地址段[DISKMAP,DISKMAX]中。据此线性关系，可根据磁盘块编号算出磁盘块的虚拟地址。

很简单，公式为 `DISKMAP + blockno * BLOCK_SIZE`

## 2.5 map_block()

**函数用法：**

```
int map_block(u_int blockno)
```

**参数：**

- blockno:即将读入内存的磁盘块号

**逻辑：**

首先用 `block_is_mapped()` 检查目标磁盘块是否已被映射入内存，没有进行映射。映射过程的本质就是为文件服务进程地址空间中的某一段申请物理内存，使用 `syscall_mem_alloc(0, disk_addr(blockno), PTE_D)`.

**注意：**

使用上面的 `alloc` 方法申请的空间为一页(PAGE_SIZE),而需要的大小为一个磁盘块(BLOCK_SIZE).

~~这三者相等!老奸巨猾的MOS啊啊啊啊!!!~~

## 2.6 block_is_mapped() （简述）

**函数用法：**

```
void *block_is_mapped(u_int blockno)
```

通过块号获取对应虚拟地址，使用 `va_is_mapped(va)` 检查是否已经被映射。是则返回这一地址( `disk_addr()` )，否则返回空指针。

- 而 `va_is_mapped()` 函数就是通过 `vpt[VPN(va)] & PTE_DIRTY` ,通过页表检测。

## 2.7 unmap_block()

**函数用法：**

```
void unmap_block(u_int blockno)
```

**参数：**

- blockno: 待操作磁盘块号

**逻辑：**

总体来说，如果目标磁盘块确实被占用，且被修改过，那么由于块缓存机制，我们在换出这个磁盘块前需先将其写回磁盘。之后在取消其映射。各步骤实现如下：

- 使用block_is_mapped,获取对应va；
- 如果满足上面两个条件（分别使用 `!block_is_free(blockno)` 和 `block_is_dirty(blockno)` 判断），调用 `write_block(blockno)` 执行写回。
- 最后调用 `syscall_mem_unmap(0,va)` ，取消文件服务进程中该页虚拟内存映射。

## 2.8 上个函数中的各个函数调用简述

### 2.8.1 block_is_free()

**函数用法**

```
int block_is_free(u_int blockno)
```

通过检查位图判断磁盘块是否为空闲，空闲返回1；被占用或者磁盘号非法返回0.（非法：磁盘号为0或超出super块中记录的磁盘号范围）

### 2.8.2 block_is_dirty()

**函数用法**

```
int block_is_dirty(u_int blockno)
```

返回 `va_is_mapped(va) && va_is_dirty(va)` 判断该块是否为'dirty'.而 `int va_is_dirty(void *va)` 中通过检查页表相应权限位 `PTE_DIRTY` 来判断是否为脏位。

**注意：** ~~一个简述的函数居然值得我写个注意，老爷们还是看看~~

如何维护一个块的脏位呢？我原本想的是维护一个数据结构，但MOS中设计的更巧妙。

磁盘块只有被调入内存，且被写入才会成为 `dirty`，那就可以借助该页在页表中的权限位进行标记咯~我们可以看看维护脏位的函数 `dirty_block()`

```
int dirty_block(u_int blockno) {
        void *va = disk_addr(blockno);

        if (!va_is_mapped(va)) { return -E_NOT_FOUND; }
        if (va_is_dirty(va)) { return 0; }
        return syscall_mem_map(0, va, 0, va, PTE_D | PTE_DIRTY);
}
```

最后一句是对标志位的修改，会重新映射。(还记得这个mem_map的作用机理吗，死去的lab2开始攻击我)

### 2.8.3 write_block()

**函数用法：**

```
void write_block(u_int blockno)
```

先调用 `block_is_mapped()` 函数确保被映射（保证写回行为不会引发错误）。然后调用 `ide_write()` 函数写磁盘。

## 2.9 dir_lookup()

**函数用法：**

```
int dir_lookup(struct File *dir, char *name, struct File **file)
```

**函数参数：**

- dir: 目录（搜寻范围）
- name: 要寻找的文件名
- file: 如果找到，将文件控制块保存在file中

**返回值：**

成功返回0，否则返回错误码

**逻辑：**

总的来说，我们需要在给定目录下寻找名字为 `name` 的文件。我们遍历目录的控制块中存储的文件控制块，比较 `f_name` 与 `name` 。该过程详解如下：

- 获取目录中的磁盘块数，并遍历。遍历时需要先将该磁盘块通过 `file_get_block(dir,i,&blk)` 读入内存。
- 遍历磁盘块下的8个文件控制块，用 `strcmp(f->f_name,name)` 判断文件名。

## 2.10 file_get_block()

**函数用法：**

```
int file_get_block(struct File *f, u_int filebno, void **blk) {
```

**参数：**

- f:指定的文件
- filebno:指定的磁盘块号（文件中的第几个磁盘块，而不是磁盘中的磁盘号）
- blk:用于存储找到的磁盘块

**返回值：**

成功返回0，否则返回错误码

**逻辑：**

此函数用于将某指定的文件中某指定磁盘块读入内存。首先调用 `file_map_blcok(f, filebno, &diskbno, 1)` 为磁盘块分配物理空间，再调用 `read_block(diskbno, blk, &isnew)` 读取磁盘块内容。

## 2.11 上述调用简述

### 2.11.1 read_block()（简述）

**函数用法：**

```
int read_block(u_int blockno, void **blk, u_int *isnew)
```

目的是确保某个磁盘块内容能被使用对应虚拟内存地址读取。首先确保磁盘号合法，以及对应磁盘块非空闲。确保读取数据有效。

之后我们要存储一个isnew的值，为调用者提供指示。调用 `block_is_mapped()` 判断，如果已经读入内存，那么是可以直接读的，不需要其他操作了，*isnew置为0.

否则置1，而且此时要调用 `syscall_mem_alloc(0, va, PTE_D)` 与 `ide_read(0, blockno * SECT2BLK, va, SECT2BLK)`，申请物理块后读入。这样，我们之后就能从该虚拟地址中读取磁盘内容啦~

最后，将va存入blk。

### 2.11.2 file_map_block()（简述）

**函数用法：**

```
int file_map_block(struct File *f, u_int filebno, u_int *diskbno,
u_int alloc)
```

调用 `file_block_walk(f, filebno, &ptr, alloc)` 遍历文件控制的磁盘块，如果找到对应的磁盘块就直接存在

`diskbno` 中，否则根据alloc控制，可以为其新申请一个磁盘块(使用alloc_block()).

# 三、文件系统的用户接口

## 3.1 open()

**函数用法：**

```
int open(const char *path, int mode)
```

**参数：**

- path: 要打开的文件路径
- mode: 操作模式

**返回值：**

返回文件描述符编号

**逻辑：**

用户进程请求打开文件时，首先将路径发送给文件服务进程(发送了一个文件描述符 `Fd fd`)；文件服务进程将文件大小，id等信息存储在该描述符中；之后用户进程根据这些信息，直接向文件服务系统请求将对应磁盘内容映射到自身的地址空间中。

上述两次通信分别调用了 `fsipc_open(path, mode, fd)` 与 `fsipc_map(fileid, i, va + i)` .这些函数都是将参数传入的信息整合在 `Fsreq_*` 等结构体中，再调用 `fsipc()` 与文件系统通信。很类似与异常处理那个味道。

而 `fsipc` 就是单纯的 `ipc_send()` 与 `ipc_recv()` 了。

## 3.2 read()

**函数用法：**

```
int read(int fdnum, void *buf, u_int n)
```

**参数：**

- funum: 文件描述符序号
- buf:保存读入内容的地址
- n:读内容最多n个字节

**返回值：**

成功时返回读取的字节数，否则返回错误码。

**逻辑：**

给定文件描述符序号后，先调用 `fd_lookup(fdnum,&fd)` 找到文件描述符；然后通过其中存储的 `fd_dev_id` ，调用 `dev_lookup(fd->fd_dev_id,&dev)` 找到要操作的外设。

之后对其中的模式进行检验（read / write）

之后调用dev中的 `r = dev->dev_read(fd, buf, n, fd->fd_offset)` 进行读入。也就是 `file_read()` ,该函数调用了 `memcpy` 进行处理。

最后，维护定位指针 `offset` 。

**注意：**

- `dev_lookup()` 的寻找方式是遍历设备表 `devtab[]`

- `fd_lookup()` 则凭借文件描述符线性排列的特点，通过页表有效性判断对应文件描述符是否被使用。从而保证有效性。

*其他内容是一些简单的接口调用了，可能后续会补充分析整个流程。~~也可能开摆~~*

# 四、文件服务过程

看来仅仅分析填空函数是不够的，我们还是把整个流程看一遍吧。

用户可以进行的文件服务有5种：
`open`，`file_close`，`file_read`，`file_write`，`file_stat`，`ftruncaye` 以及 `remove`，我们一一分析。

## 4.1 open

用户端调用函数 `int open(const char *path, int mode)`.

- 第一步：获取文件描述符。

  上面分析过，用户进程与文件服务进程通信的信息载体是文件描述符。所以要获取文件描述符。

  ```
  try(fd_alloc(&fd));
  ```

  原理是遍历页表找到首个未使用的文件描述符。

- 第二步：向文件服务进程发送open请求。

  接下来就要进行第一步通信了，我们通过：

  ```
  try(fsipc_open(path, mode, fd))
  ```

  进行通信，该函数构造一个 `Fsreq_open` 的结构体，将信息记录在结构体中，然后调用 `fsipc(FSREQ_OPEN, req, fd, &perm)`.这是用户进程进行文件服务请求的统一接口。这里就进行了一次通信的两个步骤：

  - 调用 `ipc_send(envs[1].env_id, type, fsreq, PTE_D)`.将请求信息发送给第二个进程
  - 通过 `return ipc_recv(&whom, dstva, perm)` 获取通信结果。

- 第三步：文件系统服务

由文件服务系统中，由函数 `void serve(void)` 分析可知，文件服务系统通过死循环，不断调用 `req = ipc_recv(&whom, (void *)REQVA, &perm)` 来接受其他进程的服务请求，接到请求后，便通过

```
func = serve_table[req];
func(whom, REQVA);
```

进行分发处理。当 `req` 为 `FSREQ_OPEN` 时，对应的函数为 `serve_open` 。于是，文件系统进程这边的服务开始。

- 第四步：serve_open

  文件服务系统执行该任务时，先申请一个 `strcut Open` 用于记录信息，还要调用 `file_create(rq->req_path, &f)` 检查文件是否存在。之后才正式调用 `file_open(rq->req_path, &f)` 打开文件；如果mode中包括 `O_TRUNC` 位，还要调用 `file_set_size(f, 0)` 将文件大小截断为0.

  以上过程有任何一步报错、调用 `ipc_send(envid, r, 0, 0)` 返回错误码。

  全部正常执行完毕，则调用 `ipc_send(envid, 0, o->o_ff, PTE_D | PTE_LIBRARY)` ,此时的 `o->o_ff` 中保存着文件指针 `struct File *f` ,文件id `f_fileid` ,设备id `f_fd.fd_dev_id` 等。

  - 第四步过程1：file_open()

    调用 `walk_path(path, 0, file, 0)` 根据传入的path，将文件指针保存在file中。该函数通过path中的name逐级寻找目录；寻找方式为使用 `dir_lookup(dir, name, &file)` .

  - 第四部过程2：file_set_size()

    如果原文件大小大于 `newsize` ，就调用 `file_truncate` ，将多出的block置0.之后调用 `file_flush` ,将文件内容回写到磁盘。

- 第五步：回到用户进程

  文件服务进程通过 `ipc_send` 发送信息，用户进程调用 `ipc_rece` 后，就可以从 `fsipc_open` 返回了。从上面一段我们可以看到，此时用户函数仅仅获得了文件控制块，只是获得了相关信息，还不能访问其内容。

  于是，我们获取f_size,f_fileid等信息，循环调用 `fsipc_map(fileid, i, va + i)` ，将文件内容映射到自身的地址空间内。

- 第六步： `fsipc_map()`

  比 `fsipc_open` 多关注了一个 `offset` 的信息，最后还是要调用 `fsipc(FSREQ_MAP, req, dstva, &perm)` .

  中间的通信过程于上面相同，接下来我们来看文件服务端的 `serve_map` 。

- 第七步：`serv_map()`

  调用`open_lookup()`检查文件是否已打开，原理为检查opentab以及检查对应页面的pp_ref。然后调用`file_get_block(pOpen->o_file, filebno, &blk)`，找到磁盘块（存储文件内容）。

到此，这个open的过程就完成了。看似麻烦，其实主要的操作就两部分。

## 4.2 close

其实有了最上面一个的介绍，下面的三个都是同理。考虑到还是存在差异，这里我们简要讲解。

与open不同的是，`close`中直接给出`fd`操作。

- 第一步：检查`dirty`

  首先调用`fsipc_dirty(fileid, i)`,检查文件是否为`dirty`，如果是不能直接关闭。

- 第二步：调用`fsipc_close(fileid)`，并使用`syscall_mem_unmap(0, (void *)(va + i))`取消映射，释放地址空间。

- 第三步：serv_close

  通信过程相同，我们直接看文件服务端。

  首先通过`open_lookup(envid, rq->req_fileid, &pOpen)`检查文件是不是打开，然后调用`file_close(pOpen->o_file)`.

- 第四步：file_close()

  作为文件服务系统，对于文件关闭操作没有什么特殊需求，只要要保证用户对文件的修改能写会磁盘，所以调用`file_flush`将文件内容写回磁盘即可。

## 4.3 read

由于文件已经在pen时被映射入自身地址空间，故此处只需要调用memcpy即可。

## 4.4 write

同上，使用memcpy即可。

注意写入时如果导致的文件大小的增加，则需要调用`ftruncate`增加文件大小。这里使用了`fsipc_set_size`。

## 4.4 remove

用户端无特殊处理，跳入文件服务端。

通过 `file_truncate(f, 0)` 与 `file_flush(f)` 将文件内容截断为0并写会磁盘。

# 四、往年题：

## 4.1 2023年

### lab5-1-exam

考察类似ide_read()的仿写，似乎很清晰。这里复习一下ide_read()的步骤

- 首先调用 `ide_wait()` 进行等待，为后续的操作字设置做准备
- 等待完成后通过多次syscall_dev_write()设置操作字，由于各次操作之间互不影响，所以无需等待
- 操作字写完后再次等待，
- 循环调用syscall_dev_read()读取目的内容

### lab5-1-extra

*注：2023年两次lab5课上的extra均见pdf中*

新引入了闪存的介绍，要求我们模拟闪存读写。看似很玄幻，其实不过是读写磁盘是满足一些约束，增加一些逻辑判断与动作罢了。

& 题目限制物理块号与逻辑块号都在0-31内，这样就没啥难点了吧……注意别把逻辑块和物理块搞混了就行。

思路：

全局增加三个数据结构：

- ssdmap[32]:闪存映射表
- ssdphybit[32]物理块位图（其实用一个32位变量记录也可以）

- statusTable[32]:物理块擦除表

我们要顺次实现 `init`，`read`，`write`，`erase` 四个函数

- `init()`

  闪存映射表全部清空，位图中所有物理块均化为可写状态

  （注：闪存表的清空就是以特殊值代表“映射不存在”的意思，不能记为0，要取一个取不到的值，如 `0xffffffff`）

- `int ssd_read(u_int logic_no, void *dst)`

  首先用映射表找到映射的物理块号,映射不存在返回 `-1`,存在则调用ide_read()读数据。

- `void ssd_write(u_int logic_no, void *src)`

  检查映射表，为空则分配一个物理块,这时要更新映射表，并调用ide_write()写入数据，然后标记为不可写；如果不为空，就清除原来映射，擦除原物理块数据（调用），然后再分配。相当于 `replace()`。

  这里要注意的是选择分配哪个物理块，根据题中要求：

  - 遍历所有可写块，选出擦除次数最小、块号最小的；
  - 如果擦除次数小于5，这就是找到的块，记为块A
  - 如果大于等于5，说明进入高负荷状态，此时我们遍历所有不可写块，找到次数最小、块号最小的块。
  - 将块B内容写入A中，ide_read() + ide_write();
  - A标记为不可写，更改B的逻辑映射至A，擦除B，B是分配的物理块（其实就是看B太闲了，让B代替A去干活了）

- `void ssd_erase(u_int logic_no)`

  检查映射表，为空则不处理，不为空，则使用memcpy清除物理块，维护擦除次数，标记为可写。

以下附上zy学姐代码：

```
u_int ssdtable[32];
u_int ssdbitmap[32];//1 可写
u_int ssdnum[32];
void ssd_init() {
    for (u_int i = 0; i < 32; i++) {
        ssdtable[i] = 0xffffffff;
        ssdbitmap[i] = 1;
        ssdnum[i] = 0;
    }
```

```
}
int ssd_read(u_int logic_no, void *dst) {
    if (ssdtable[logic_no] == 0xffffffff) {
        return -1;
    }
    ide_read(0, ssdtable[logic_no], dst, 1);
    return 0;
}
void ssd_write(u_int logic_no, void *src) {
    if (ssdtable[logic_no] != 0xffffffff) {
        ssd_erase(logic_no);
    }
    //alloc----------------------
    u_int physical_no = 0xffffffff;
    for (u_int i = 0; i < 32; i++) {
        if (ssdbitmap[i] == 1) {
            if (physical_no == 0xffffffff) {
                physical_no = i;
            } else {
                if (ssdnum[i] < ssdnum[physical_no])
                    physical_no = i;
            }
        }
    }
    if (ssdnum[physical_no] >= 5) {
        u_int help_no = 0xffffffff;
        u_int help_logic = 0xffffffff;
        for(u_int i = 0; i < 32; i++) {
            if (ssdbitmap[i] == 0) {
                if (help_no == 0xffffffff) {
                    help_no = i;
                } else {
                    if (ssdnum[i] < ssdnum[help_no])
                        help_no = i;
                }
            }
        }
        for (u_int i = 0; i < 32; i++)
            if (ssdtable[i] == help_no) {
                help_logic = i;
                break;
            }
        u_int help_data[128];
        if (ssd_read(help_logic, (void *)help_data) != 0)
            user_panic("wrong in ssd_write's help_data\n");
        ide_write(0, physical_no, (void *)help_data, 1);
```

```
            ssdbitmap[physical_no] = 0;
            ssd_erase(help_logic);
            ssdtable[help_logic] = physical_no;
            physical_no = help_no;
        }
        //----------------------------
        ssdtable[logic_no] = physical_no;
        ide_write(0, physical_no, src, 1);
        ssdbitmap[physical_no] = 0;
}
void ssd_erase(u_int logic_no) {
        if (ssdtable[logic_no] == 0xffffffff) {
            return;
        }

        //all 0 in 物理块------------
        u_int zero[128];
        for (u_int i = 0; i < 128; i++)
            zero[i] = 0;
        ide_write(0, ssdtable[logic_no], (void *)zero, 1);
        //----------------------------
        ssdnum[ssdtable[logic_no]]++;
        ssdbitmap[ssdtable[logic_no]] = 1;

        ssdtable[logic_no] = 0xffffffff;
}
```

代码与我的思路基本吻合。学姐在擦除中使用的构造全0块并使用 `ide_write()` ，也是好思路.

## lab5-2-exam

基本是照抄原系统中的open过程，增加一个open_at过程，实现寻找指定目录下给定相对路径的文件并打开。记得抄完整个过程。

## lab5-2-extra

*考在了 `fsformat.c` 上我擦*

主要涉及两处修改

- 增加 `write_symlink()` 函数，调用库函数 `readlink()` 函数获取链接路径并保存，之后设置目标文件大小类型
- 在open函数中增加对链接类型的处理，事实上就是递归调用

参考的代码如下：

```
//tools/fsformat.c
void write_symlink(struct File *dirf, const char *path) {
    int iblk = 0, r = 0, n = sizeof(disk[0].data);
    struct File *target = create_file(dirf);
    char targetpath[2048] = {0};
    int len = readlink(path, targetpath, 2047);
    /* in case `create_file` is't filled */
    memcpy(disk[nextbno].data, targetpath, len);
    disk[nextbno].data[len]='\0';

    // Get file name with no path prefix.
    const char *fname = strrchr(path, '/');
    if (fname) {
        fname++;
    } else {
        fname = path;
    }
    strcpy(target->f_name, fname);

    target->f_size = 2048;
    target->f_type = FTYPE_LNK;

    save_block_link(target, 0 , next_block(BLOCK_DATA));
}

//user/lib/file.c
int open(const char *path, int mode) {
    //......略

    if(ffd->f_file.f_type == FTYPE_LNK){
        return open(fd2data(fd), mode);
    }else{
        return fd2num(fd);
    }
}
```

lab5-2让我认识到，第一版的实际只包含了lab5一半的内容，fsformat和文件服务接口相关内容摆不了一点。