LAB5实验报告

思考题

thinking5.1

> 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。这是 一种错误的行为，在实际编写代码的时候这么做会引发不可预知的问题。请思考：这么做 这会引发什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存更新的策略来考虑。

这是由于缓存到Cache中后不会实际写入设备，也就无法触发设备的操作。如写入时钟设备的刷新位置意图刷新时钟时，如果写入了cache中则读取到的仍然是之前的时钟。

关键在于对于内存的读写是唯一且直接改变内存的方法，因此可以经过Cache，而对设备的读写只是设置设备运行的方式，因此不能用Cache进行加速。这主要是由于Cache的更新策略包括写直通和写回，写直通可以满足设备的写入要求，而写回则不行。并且Cache由于是为内存设计，因此默认只要不写入cache缓存的位置主存中的内容就不会改变，这与设备的工作原理不符。

当使用不同设备时，会有不同的表现，当设备起到类似内存的作用时，可以正常完成。而如果涉及到设备自我更新导致的数据不一致，则无法正常完成操作。

thingking5.2

> 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制 块？一个目录下最多能有多少个文件？我们的文件系统支持的单个文件最大为多大？

`#define BY2FILE 256``和` `#define BY2BLK BY2PG` 决定了每个文件控制块256字节大小，每块一页大小，每页4096字节大小，故每块能保存16个文件控制块。

`#define MAXFILESIZE (NINDIRECT * BY2BLK)` 定义了文件大小为NINDIREC*BY2BLK = 1024*4096 B=4MB大小。

一个目录下最多4MB大小的文件控制块，既16384个文件。

thinking5.3

由于我们的缓存块放置在DISKMAP到DISKMAX之间且与物理块一一对应，共有0x40000000字节，既1G大小。

thinking5.4

> 在本实验中，fs/serv.h、user/include/fs.h 等文件中出现了许多宏定义， 试列举你认为较为重要的宏定义，同时进行解释，并描述其主要应用之处。

1. fs/serv.h
    1. PTE_DIRTY 定义了脏位的位置，从而用与运算进行计算。
    2. DISNNO 定义了硬盘号，供系统调用设置硬盘号使用。
    3. BY2SECT 磁盘扇区大小。
    4. SECT2BLK 每块中的扇区数量。
    5. DISKMAP 定义了缓存块保存位置
    6. DISKMAX 定义了缓存区大小上限
2. user/include/fs.h
    1. BY2BLK define the bytes number of block
    2. BIT2BLK define the bits number of block
    3. MAXNAMELEN define the max len of file name. Used to check and set the file name
    4. MAXPATHLEN define the max len of pathname, Used to check and set the path name
    5. NDIRECT define the direct index block number. While NINDIRECT define the indirect index number of block.
    6. FS_MAGIC define the magic number in SuperBlock.

thinking 5.5

> 在 Lab4“系统调用与 fork”的实验中我们实现了极为重要的 fork 函数。那 么 fork 前后的父子进程是否会共享文件描述符和定位指针呢？请在完成上述练习的基础上 编写一个程序进行验证。

从一次

会复制当前已有的文件描述符和指针，其中一方更新后另一方不会更新，这是由于COW范围内的数据会在fork后访问时复制一份，故而会在fork时共享所有指针，但是对于每个进程而言都时独立的。

验证程序基于fstest.c，在open检查后添加fork和相关检查，结果如下：

```
FS is running
superblock is good
read_bitmap is good
open is good
This is a different message of the day!
This is a different message of the day!
read is good
panic at fstest.c:35: read returned wrong data
[00001802] destroying 00001802
[00001802] free env 00001802
i am killed ...
open again: OK
read again: OK
file rewrite is good
file remove: OK
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
panic at sched.c:46 (schedule): 'TAILQ_EMPTY(&env_sched_list)' returned
```

代码如下：

```
15          fdnum = r;
16          debugf("open is good\n");
17
18          if ((n = read(fdnum, buf, 511)) < 0) {
19                  user_panic("read /newmotd: %d", r);
20          }
21                  debugf("%s", buf);
22          if (strcmp(buf, diff_msg) != 0) {
23                  user_panic("read returned wrong data");
24          }
```

```
25          buf[0]='\0';
26          seek(fdnum, 0);
27          int forkr = fork();
28          if(forkr == 0){
29                  //seek(fdnum,0);
30                  if ((n = read(fdnum, buf, 511)) < 0) {
31                          user_panic("read /newmotd: %d", r);
32                  }
33                  debugf("%s", buf);
34                  if (strcmp(buf, diff_msg) != 0) {
35                          user_panic("read returned wrong data");
36                  }
37                  debugf("children read same file is same!\n");
38                  return 0;
```

fork前设置文件指针到0，fork后父子进程均读取，发现父子只有一个进程可以读取成功，这说明父子进程的文件描述符是共享的。

thinking5.6

> 请解释 File, Fd, Filefd 结构体及其各个域的作用。比如各个结构体会在哪 些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。

```
35 struct Fd {
36          u_int fd_dev_id;
37          u_int fd_offset;
38          u_int fd_omode;
39 };
```

- fd_dev_id 时file所属devid，filedev中dev_id是 `f` ，定义在 `file.c` 。
- fd_offset是当前file的指针位置。
- omode是文件操作模式，在open时设定。

与文件对应，用来进行文件读写信息存储

```
50 struct Filefd {
51          struct Fd f_fd;
52          u_int f_fileid;
```

```
53          struct File f_file;
54 };
```

- f_fd是文件描述符
- f_fileid是文件的id
- f_file是文件控制块

与文件对应，用来同时记录读写相关信息和文件控制信息

```
26 struct File {
27          char f_name[MAXNAMELEN]; // filename
28          uint32_t f_size;         // file size in bytes
29          uint32_t f_type;         // file type
30          uint32_t f_direct[NDIRECT];
31          uint32_t f_indirect;
32
33          struct File *f_dir; // the pointer to the dir where this fi
    memory.
34          char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizec
35 } __attribute__((aligned(4), packed));
```

分别是文件名、文件大小、文件类型、直接索引、间接索引块号，父目录dir，填充区

与文件对应，用来记录所有文件信息，其中直接索引存储块号，间接索引存储保存简介索引的块的块号。

物理块与缓存块一一对应。

thinking5.7

> 图5.7中有多种不同形式的箭头，请解释这些不同箭头的差别，并思考我们 的操作系统是如何实现对应类型的进程间通信的。

ENV_CREATE(user_env)和ENV_CREATE(fs_serv)用来启动相关进程，具体操作在lab3中已有。

fs_serv中调用init后进入服务循环，不断接受ipc并进行服务

user中调用open后分配fd，调用fsipc_open()，最终构造请求调用fsipc，通过ipc_send(fsreq)向处于服务循环中的fs_serv发送请求，fs_serv完成操作后通过ipc_send()向调用者返回结果。

值得注意的是ipc_recv对syscall_ipc_recv进行了包装，在完成接受后保存whom和perm到指定位置。

同样的，ipc_send进行包装，如果没有找到接收者会放弃cpu，并循环尝试。由此，通过轮询实现了多个进程向一个进程发送信息。

实验难点

> fs文件夹下，ide.c包含硬盘ide操作，fs.c包含文件系统相关函数，serv.c包含服务ipc请求的整个进程，对文件系统进行封装。

fs.c (./fs/fs.c) （100%）

关键路径：

- int file_create(char *path, struct File **file)
  从该函数触发，调用了该文件的大部分函数
  目标：创建path对应的文件，设置file指向创建的文件
  - walk_path实现文件查找，如果已经能找到对应的文件了，也就不需要再创建了。
  - 如果出现了其他错误或者压根没找到对应的dir，则返回r。由于在walk_path中保证了dir只有最后一级才会设置，因此这里相当于检查了除最后文件外路径完整。
  - 调用dir_alloc_file()在找到的最后一级创建文件。
- int walk_path(char *path, struct File **pdir, struct File **pfile, char *lastelem)
  目标：从根目录开始按照path寻找path的最后一个文件/目录
  当找到对应文件时，正常设置pfile和pdir。
  当找到了最后一级文件夹时，会设置pdir为最后一级文件夹。
  否则不设置，pdir为0。由此传输给调用者调用结果，表明没有找到最后一级。
  - 初始化：
    - 预处理path，跳过斜杠
    - 设置初始file指向root
    - 设置初始name为0

        - 设置初始pdir为0
        - 设置初始pfile为0
    - 执行遍历，直到path指向末尾 **\0**
        - 设置dir为当前file，p记录当前path，跳过文件/目录名使path指向下一个斜杠/0。
        - 检查记录的文件名长度
        - 将下一级的文件名拷贝到name
        - 跳过path的下一个斜杠
        - 如果不是目录，则找不到文件，返回错误
        - 如果是目录，调用dir_lookup(dir,name,&file)
            - 如果找不到，且目录已经空了，则设置pdir为当前dir，复制name到laste，设置pfile为0，返回错误。此时依然有一定作用，因为保存了最后找到的dir和name，可以供file_create使用。
            - 如果找到了，则继续
    - 返回结果
        - 此时，最后一次遍历找到了file，设置pfile为file1，并设置pdir为dir
- int dir_lookup(struct File *dir, char *name, struct File **file)
  目标：在dir中寻找文件名为name的文件，并保存在file中
    - 首先获得dir中的文件块数量
    - 遍历文件块，调用file_get_block使blk指向第i个缓存块，File*files指向这个块，遍历其中文件。
        - 调用strcmp，如果文件名与目标name相等，则设置file为对应f，同时设置f的f_dir为当前dir。
    - 如果找不到，返回错误。
- int file_get_block(struct File *f, u_int filebno, void **blk)
  目标：得到f中第filebno号的文件块，并使blk指针指向它。
    - 首先使用file_map_block获得diskbno，允许自动创建一级索引。
    - 使用readblock读取diskbno对应的文件块，保存缓存块va到*blk
- int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc)
  目标：在diskbno保存文件中第filebno块对应的物理块号
    - 首先调用file_block_walk()得到指向**向物理块的映射的项**的指针ptr

    - 如果ptr指向的项对应物理块是0，则不存在这个块，则检查是否创建，如果创建则创建键一个，并保存到文件中的项里，否则返回错误。
    - 设置diskbno为对应的物理块号
- int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int alloc)

  目标：找到 **指向**文件中**保存第filebno块的物理块号**的指针，保存到*ppdiskbno。
    - 检查filebno，如果太大则返回错误
    - 直接索引，直接返回
    - 间接索引，如果不alloc且未建立，则返回错误，否则保证完成后间接索引存在且返回正确的槽位。在此过程中，调用alloc_block()和read_block创建、读取正确的索引。
- int alloc_block(void)

  目标：创建一个block，首先在bitmap中找空块，然后在内存中分配空间。
    - 寻找blockid号，调用alloc_block_num()
    - 分配空间，调用map_block()，在blockid对应的缓存块处分配空间。
- int alloc_block_num(void)

  目标：找到第一个对应空物理块的物理块id号
    - 跳过boot、super和bitmap块，从第四个物理块开始遍历，找到第一个空块并返回id。
- int map_block(u_int blockno)

  目的：在blockno对应的缓存块位置**分配**一页内存， **不实际映射**。
    - 首先判断blockno对应的block是否已经被映射。调用block_is_mapped()由于正常地址返回值大于零，可以当作true判断。
    - 如果没有映射，则调用syscall_mem_alloc在虚拟地址diskaddr(blockno)处分配一页空间
- void *block_is_mapped(u_int blockno)

  目标：检查blockno对应的物理块是否已经在内存中缓存
    - diskaddr获取blockno对应物理块的va，调用va_is_mapped检查是否映射。
    - 由于缓存块和物理块一一对应，故而可以如此检查。
    - 返回值是va，而va在用户区大于0。
- int read_block(u_int blockno, void **blk, u_int *isnew)

  目标：blk指向blockno对应的缓存块

    - 如果super分配了，且blockno大于super中的设置，则读取block超范围，panic。
    - 如果bitmap初始化了，且对应的block没有使用，panic
    - 如果该block已经缓存，则保存va到blk。
    - 如果该block没有缓存，调用syscall_mem_alloc分配页面，并调用ide_read读取页面。
- int dir_alloc_file(struct File *dir, struct File **file)
  目标：在dir中分配空文件，并让file指向它
    - 遍历dir中的block，遍历每个block中的file，找到第一个空file，使*file指向它。
    - 如果没有找到任何空file，则新创建一个block，使*file指向其中的第一个file。
- int va_is_dirty(void *va)
    - 返回vpt中对应va的项是否有PTE_DIRTY位
- int block_is_dirty(u_int blockno)
  目标：判断blockno对应的块是否已经污染
    - 首先调用va_is_mapped判断是否已经分配了blockno对应缓存
    - 随后调用va_is_dirty检查是否已经污染
- int dirty_block(u_int blockno)
  目标：标记blockno污染
    - 如果缓存未分配或者已经污染，则返回错误或者0；
    - 否则调用syscall_mem_map，设置自身的vpt中对应的blockno的PTE_DIRTY位
- void write_block(u_int blockno)
  目标：向磁盘中写入blockno对应块
    - 如果没有映射，panic
- void file_flush(struct File *f)
    - 根据file的大小，遍历每一块，判断是否已经映射，如果映射了且dirty则写入，否则跳过该块。
- void fs_sync(void)
    - 相当于对于整个文件系统的flush，遍历super块中定义的所有块，如果dirty则写入。
- void unmap_block(u_int blockno)
    - 判断blockno是否已经缓存

    - 如果已经使用且污染则向磁盘中写入，在这里block_is_dirty中检查了是否已经分配，故不需要再本函数中体现
    - 调用syscall_mem_unmap，取消映射。
- fs_init()*
    - 读super块，检查正确性，设置super变量为super块
    - 尝试写入，检查正确性，设置super变量为super块
    - 读bitmap块

ide.c（./fs/ide.c）（100%）

- void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
    - 按照guidebook中的读取流程，使用syscall按流程读取。
- void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
    - 同理

serv.c (./fs/serv.c) (100%)

当系统添加了这个ENV后，分三步进行初始化

- serve_init()
    - 初始化OPEN表
        - OPEN表保存了文件描述符和文件的对应关系、文件id和打开时设置的操作模式，包含一个File结构体指针和一个Filefd结构体指针。
        - 在初始化时，opentab是已经定义并初始化完成的Open结构体数组，共MAXOPEN=1024长
        - 遍历opentab中的每个结构体，设置fileid为在opentab中的索引下标，设置filefd保存位置为从FILEVA=0x60000000开始的第fileid个页面。
- fs_init()
    - 在fs.c文件中
- serve()
    - 开始进入服务循环，不断接受服务。
- void serve_open(u_int envid, struct Fsreq_open *rq)
    - 对应FSREQ_OPEN请求

    - 首先调用open_alloc在opentab中申请位置，申请后o指向opentab中的一个空位置。
    - 调用file_open()找到req中path的对应的文件，f代表该文件。
    - 设置ff指向o的o_ff空间
    - 设置ff中的file为打开的f
    - 设置ff的fileid为o的fileid，设置omode、设置devid为devfile的devid
    - 发送结果，映射o->o_ff页面到用户进程，并设置权限位为PTE_D和PTE_LIBRARY，表明可写且父子进程共享。
- int open_alloc(struct Open **o)
    - 遍历opentab，检查opentab每个结构体的off对应页面被使用了几次。
        - 如果没被使用，说明还未分配对应的页面，则调用syscall_mem_alloc在该位置申请一个PTE_D|**PTE_LIBRARY** 标记的页面，这使得fork时父子进程共享该位置保存的filefd结构体。
        - 无论是0次还是1次，都说明该位置可以被使用，这是由于在serveOpen中文件系统服务进程会将o->o_ff地址作为发送页面发送给接收进程，由此会有两个进程的地址映射到这同一个页面，因此任何还在使用的Open结构体的ff地址对应的引用次数都是大于等于2的。通过对0次和1次的判断，可以仅在需要的时候进行内存申请。
        - 这里的fileid每次打开时均加MAXOPEN，保证了id的唯一性。
        - 找到空打开opentab位置后，设置o指向这里，并清空o_ff位置。
- void serve_map(u_int envid, struct Fsreq_map *rq)
    - 目标：找到envid对应的进程中rq中filedid对应文件，映射其offset位置对应块。
    - 找到envid中的fileid打开文件。计算req中offset对应的文件块。用file_get_block在文件中找到对应块，并作为srcva返回，映射到调用者需要的地方。同样，这里的权限是PTE_D | PTE_LIBRARY。
- int open_lookup(u_int envid, u_int fileid, struct Open **po)
    - 目标：找到envid中fileid对应的打开的文件
    - 由于分配时保证了fileid是open表索引加整数倍的MAXOPEN，故只需要取余MAXOPEN即可找到fileid对应的open结构体，判断o_ff对应的页面引用次数，如果只有1次或者fileid不匹配，则说明该打开文件已经无效，返回错误。否则，设置*po指向找到的o。
    - 值得注意，这里没有用到envid，可能作为考察点。

- void serve_set_size(u_int envid, struct Fsreq_set_size *rq)
  - 目标：设置文件大小
  - 首先查open结构体，调用file_set_size()设置文件大小。
  - 返回成功值0或者错误值。
- void serve_close(u_int envid, struct Fsreq_close *rq)
  - 目标：关闭对应的文件。
  - 首先查open结构体，调用file_close()关闭文件。
  - 返回成功值0或者错误值。
- void serve_remove(u_int envid, struct Fsreq_remove *rq)
  - 目标：删除文件
  - 调用file_remove，返回其返回值。
- void serve_dirty(u_int envid, struct Fsreq_dirty *rq)
  - 目标：
  - 查找open结构体
  - 调用file_dirty()设置file对应offset为dirty
  - 返回成功/错误值
- void serve_sync(u_int envid)
  - 调用fs_sync()，同步。
  - 返回成功

> user/lib目录下，file对于文件类型的设备 相关操作进行了定义从而组成devfile表示文件系统设备，fsipc包含对文件操作ipc调用的封装，ipc定义了用户态的ipc操作，对发送接收进行了封装，发送可以阻塞发送，接受可以保存参数到指定地址。fd包含了fd结构体及相关定义使用。

file.c（100%）

用处：定义了一个Dev叫做devfile，并提供给fd.c调用。

- int ftruncate(int fdnum, u_int size)
  - 目标：调整文件size到指定的size
  - 如果size不满足要求、或者fdnum对应的fd不存在、或者fdnum对应的fd不是文件，则返回异常值。
  - 否则，初始化局部变量f为Filefd，提取fileid、oldsize，设置新size

    - 调用fsipc_set_size()，调整文件大小。
    - 向上取整oldsize，map新区域到内存，如果失败，则设置回oldsize
    - 向上取整size，unmap需要释放的内存
- int open(const char *path, int mode)
    - 目标：打开一个path，以mode为操作模式
    - 调用fd.c中的fd_alloc(&fd)向fd中存储该进程**fd表**中第一个空的地址。
    - 调用fsipc_open(path,mode,fd)，打开path，并保存信息到fd中。巧妙设计是fdalloc中的位置对应的是一页一页的空间，与ipc的dstva页面映射做对应，从而用fd的空间接受来自文件系统的ff块，故fd保存的实际上就是Filefd，因此可以在来回转换。
    - 打开了之后分别设置va为文件内容保存地址，ffd为Filefd指针，size为文件大小，fileid为文件id。
    - **注意**，此时遍历文件size对应的所有块，向va地址写入整个文件，且调用的是serv.c中的map，权限是LIBRARY，父子进程共享。
    - 最后，返回fd编号
- int file_close(struct Fd *fd)
    1. fd转为filefd，获取fileid和size，找到保存文件内容的位置va
    2. 遍历文件内所有块，调用fsipc强行设置为dirty
    3. 调用fsipc_close，刷新该id的文件内容到磁盘。
    4. 遍历文件每个块，调用syscall_mem_unmap取消地址映射，释放这些块
    5. **注意**！！！ 这里出现了函数重名，在此处是供所有用户进程调用的file_close，而serv.c中调用的是fs的file_close。
- static int file_read(struct Fd *fd, void *buf, u_int n, u_int offset)
    - 获取到文件大小
    - 切割溢出的大小，调用memcpy拷贝问文件内容到buf
    - 返回实际拷贝的字节数n
- int read_map(int fdnum, u_int offset, void **blk)
    - 用fdnum找到fd，判断是不是一个文件系统文件描述符
    - 计算va为文件内容地址，判断va+offset对应的位置是否已经读入内存，如果是的话，设置*blk为va+offset
- static int file_write(struct Fd *fd, const void *buf, u_int n, u_int offset)
    - 目标：向文件偏移offset处写入buf中的n个字节。

    - 计算tot为写入位置上限，如果大于当前大小，调用ftruncate修改文件大小。
    - 向文件中写入对应的数据。
- static int file_stat(struct Fd *fd, struct Stat *st)
    - 目标：获取文件状态
    - 首先把fd转化为Filefd
    - 拷贝文件名、文件大小、文件类别到st。

fsipc.c

包装fs的ipc调用

- int fsipc_open(const char *path, u_int omode, struct Fd *fd)
    - 检查path长度
    - 向请求缓存中写入path、omode，调用fsipc，分别传入FSREQ_OPEN、req、fd、&perm，表明这是一个OPEN请求，内容是req，文件描述符地址是fd并作为dstva，保存返回值到perm中。
    - 其它ipc包装类似。
- u_char fsipcbuf[BY2PG] __attribute__((aligned(BY2PG)))
    - 供文件系统ipc请求构造使用的缓存
- static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm)
    - 调用ipc_send()向fserver对应的envid发送type、fsreq和PTE_D
    - 调用ipc_recv得到返回值，写入传输的页面到dstva

ipc.c （100%）

包装优化的ipc操作函数。

- void ipc_send(u_int whom, u_int val, const void *srcva, u_int perm)
    - 只要没有接收者，就一直尝试发送。每次尝试后yield
- u_int ipc_recv(u_int *whom, void *dstva, u_int *perm)
    - 调用syscall_ipc_recv(dstva)，接收页面。并维护whom和perm，env结构体在syscall_ipc_recv中保存的发送者和perm。

fd.c

该文件中包装了对于fd可以进行的相关操作，通过Dev中存储的函数可以进一步调用各个设备中的操作函数。

- int fd_alloc(struct Fd **fd)
  - 利用INDEX2FD，遍历第0个到第MAXFD-1（31）个fd表位置，检查vpd对应位置是否有效，如果无效，说明是空的，则把此fd位置分配。如果vpd有效，vpt无效，同样表明该fd位置是空的，分配此位置。
  - 如果在fd表中找不到空fd位置，则返回-E_MAX_OPEN
- #define INDEX2FD(i) (FDTABLE + (i)*BY2PG)
  - 从FDTABLE开始，每页保存一个fd，由此算出第i个fd保存位置
- #define FDTABLE (FILEBASE - PDMAP)
  - 定义fd表位置为FILEBASE下的4MB空间，即1024页
- #define FILEBASE 0x60000000
  - 这里定义了FILEBASE的位置，可以这样说，filebase下的一页中有32个fd，filebase上的每4MB保存着一个文件的内容。且fd和文件内容按顺序一一对应。
- int read(int fdnum, void *buf, u_int n)
  - 往buf中读fdnum对应的文件指针之后的n个字节
- int readn(int fdnum, void *buf, u_int n)
  - 不断循环，直到读够n个字节。
- int write(int fdnum, const void *buf, u_int n)
  - 从fdnum对应的fd指针开始写入buf中的前n个字节
- int seek(int fdnum, u_int offset)
  - 调整fdnum对应的fd的指针位置。
- int fstat(int fdnum, struct Stat *stat)
  - 获取fdnum对应的设备状态
- int stat(const char *path, struct Stat *stat)
  - 获取path对应的文件的状态。

实验感想

本单元实验难度大、内容多，需要对于文件系统有全面的了解，并对于具体实现有深入的理解。

对于本单元的学习，需要从阅读代码触发，理解代码的含义和在整个系统调用链中的作用，才能理解系统如何运作。要求较高的代码阅读能力。

比较精妙的设计是fs.c中对于各种操作的包装，通过一层层的抽象包装将复杂的操作拆分为不同函数的相互调用，体现了面向过程设计的抽象技巧。