

## 实验思考题

### Thinking 5.1

- 查阅资料，了解 Linux/Unix 的 /proc 文件系统是什么？

/proc 文件系统是一个虚拟文件系统，通过它可以使用一种新的方法在 Linux 内核空间 and 用户空间之间进行通信。在 /proc 文件系统中，我们可以将对虚拟文件的读写作为对内核中实体进行通信的一种手段，与普通文件不同的是，这些虚拟文件的内容为动态创建。/proc 目录下的文件大小都是 0，因为这些文件本身并不存在于硬盘中，且文件本身并不是一个真实的文件，只是一个接口。当我们去读取该文件时，其实内核并不是去硬盘上读取这些文件，而是映射为内核内部的一个数据结构被读取，并且格式化为字符串返回。因此尽管我们感觉 proc 文件与普通的文件一样，但是实际上文件的内容是实时从内核中数据结构中返回而来的。

- 有什么作用？

在 Linux 内核空间和用户空间之间进行通信。

用户可以通过这些文件查看有关的硬件以及当前进程的信息，甚至通过改变其中文件来改变内核运行状态。

- Windows 操作系统又是如何实现这些功能的？

在 Windows 系统中，通过 Win32 API 函数调用来完成与内核的交互。

- proc 文件系统的设计有哪些好处和不足？

- 好处：proc 文件系统的设计将对内核信息的访问交互抽象为对文件的访问修改，简化了交互过程。
- 不足：Linux，只有一个根目录 /，所有东西都是从这开始。

### Thinking 5.2

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。

kseg0 是存放内核的区域，一般通过 Cache 访问。

对于写入操作：

在采用 Write-back 刷新策略时，写入数据只有在 Cache 被换出时才会进行写回，导致后面的操作覆盖了前面操作，只进行最后一次操作。对串口设备，只有 Cache 刷新后才能看到输出，且只能看到最后一个字符。类似的，IDE 磁盘可能只会写入最后一个扇区。

对于读取操作：问题更大，任何一种策略都可能会读取到旧的、过时的数据，因此产生错误。

### Thinking 5.3

比较 MOS 操作系统的文件控制块和 Unix/Linux 操作系统的 inode 及相关概念，试述二者的不同之处。

inode 只保存文件名和指针，但是 MOS 中的 File 结构体保存了文件名、文件大小、文件类型、文件所在目录和指针。

inode 只有间接指针，但是 MOS 中的 File 结构体中 f\_direct[NDIRECT] 保存了直接指针。

inode 中有多级指针，但是 MOS 中只有一级指针。

## Thinking 5.4

- 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？

```

1  #define BY2PG      4096
2  #define BY2BLK     BY2PG
3  #define BY2FILE    256
4  struct File {
5      u_char f_name[MAXNAMELEN];    // filename
6      u_int f_size;                  // file size in bytes
7      u_int f_type;                  // file type
8      u_int f_direct[NDIRECT];
9      u_int f_indirect;
10
11     struct File *f_dir;             // the pointer to the dir where this file is
in, valid only in memory.
12     u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
13 };
14 #define FILE2BLK (BY2BLK/sizeof(struct File))

```

根据上面的节选， $\text{sizeof}(\text{struct File}) = \text{BY2FILE} = 256\text{B}$ ， $\text{BY2BLOCK} = 4096\text{B}$ ，则一个磁盘块最多存储  $\frac{4096\text{B}}{256\text{B}} = 16$  个文件控制块。

- 一个目录下最多能有多少个文件？

一个目录（File 结构体）最多指向 1024 个磁盘块，一个磁盘块中有 16 个文件控制块，一个目录下，最多有  $1024 \times 16 = 16384$  个文件。

- 我们的文件系统支持的单个文件最大为多大？

File 结构体中，f\_indirect 指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针，每个指针的大小是 4，所以一个磁盘块可以存储 1024 个指针。其中前 10 个指针没有使用，但是另有 f\_direct[NDIRECT] 指向 10 个磁盘块。所以单个文件最多由 1024 个磁盘块构成，大小是  $1024 \times 4096 = 4\text{MB}$ 。

## Thinking 5.5

请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最大磁盘大小是多少？

```

1  /* Maximum disk size we can handle (1GB) */
2  #define DISKMAX    0x40000000

```

代码里写了，1GB

## Thinking 5.6

如果将 DISKMAX 改成 0xC0000000，超过用户空间，我们的文件系统还能正常工作吗？为什么？

显然不能，要是能超过为什么不用，哪有有空间能用还不用道理，因为如果这样设计，写磁盘的过程中会覆盖掉内核的关键内容，可能会令操作系统运行异常。

## Thinking 5.7

在 lab5 中，fs/fs.h、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，并进行解释，写出其主要应用之处。

```

1  #define NBLOCK 1024 // 一块磁盘里面的block数目
2  uint32_t nbitblock; // 用于存储bitmap的block数目
3  uint32_t nextbno; // 下一块可用的block编号
4
5  struct Super super; // 超级块
6  struct Block {
7      uint8_t data[BY2BLK];
8      uint32_t type;
9  } disk[NBLOCK];
10
11 // 下面的内容在include里面的fs.h里面
12 #define BY2BLK BY2PG // 一个block对应的字节，也就是说，一个block正好等于一页大小
13 #define BIT2BLK (BY2BLK * 8) // 一个block对应的*位数*，一字节等于 8 位，所以要乘 8
14
15 #define MAXNAMELEN 128 // 用于存文件名的 char 数组大小，由于最后一个必定为'\0',所以只能
    存 127 个字符
16 #define MAXPATHLEN 1024 // 和上面的类似，只不过是用来存路径的
17
18 #define NDIRECT 10 // 直接引用的个数，可以认为是 10 个指针，这里用 int 存 block 的下标
    来代替了指针的作用
19 #define NINDIRECT (BY2BLK / 4) // 间接引用块的指针个数，由于一个int是 32 位，也就是
    4byte，所以是除 4
20
21 #define MAXFILESIZE (NINDIRECT * BY2BLK) // 最大文件大小，那么就是引用指针的个数 * 一块
    大小
22
23 #define BY2FILE 256 // 定义了一个 File 结构体（文件索引结构体）所占用的大小
24
25 struct File { // 文件索引结构体
26     u_char f_name[MAXNAMELEN]; // 文件名
27     u_int f_size; // 文件大小（字节），其实就是引用块所占的总大小
28     u_int f_type; // 文件类型，有两种，一种是普通文件，一种是目录
29     u_int f_direct[NDIRECT]; // 直接引用指针
30     u_int f_indirect; // 指向间接引用块

```

```

31
32     struct File *f_dir;    // 指向包含这个文件的目录，只在内存中存在着（真正写入文件之后没有这
    部分）
33     u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4]; // 为了填满
    BY2FILE，而开了这部分内容
34 };
35
36 #define FILE2BLK (BY2BLK / sizeof(struct File)) // 就是一个 block 能容纳多少个
    file 索引
37
38 // File types
39 #define FTYPE_REG    0    // 普通文件
40 #define FTYPE_DIR    1    // 目录
41
42
43 // File system super-block (both in-memory and on-disk)
44
45 #define FS_MAGIC    0x68286097    // 神秘代码，用来表示这是个合法的文件，致敬了属于是
46
47 struct Super {
48     u_int s_magic;    // Magic number: FS_MAGIC
49     u_int s_nblocks;    // Total number of blocks on disk
50     struct File s_root;    // Root directory node
51 };

```

## Thinking 5.8

阅读 user/file.c，你会发现很多函数中都会将一个 struct Fd \* 型的指针转换为 struct Filefd \* 型的指针，请解释为什么这样的转换可行。

```

1 struct Filefd {
2     struct Fd f_fd;
3     u_int f_fileid;
4     struct File f_file;
5 };

```

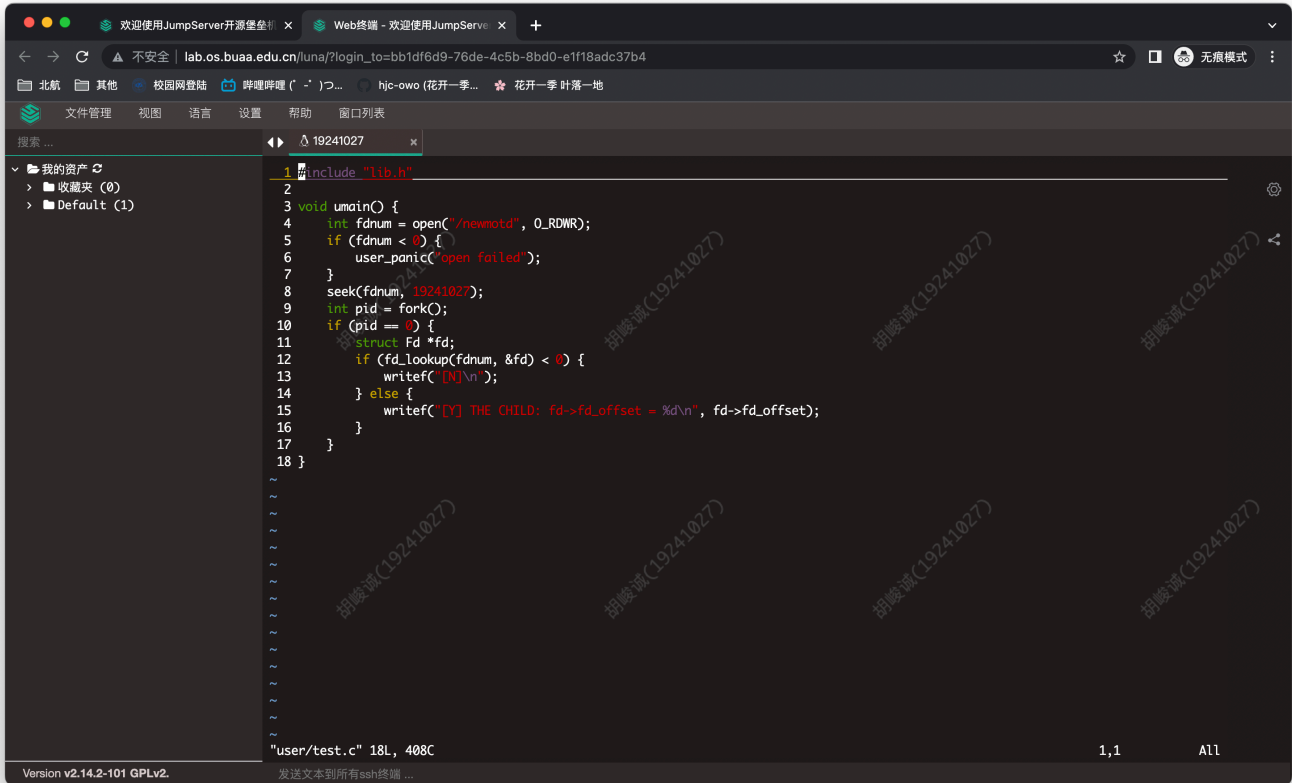
C 语言中指针指向的是一个地址，只要能取出合法的元素，就可以在各种类型之间相互进行转化。因为在结构体 Filefd 中储存的第一个元素就是 struct Fd \*，他们的指针实际上指向了相同的虚拟地址，所以可以通过指针转化访问 struct Filefd 中的其他元素。

## Thinking 5.9

在 lab4 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。

理论上来说会，但是实际上也确实 `fork` 前后的父子进程会共享文件描述符和定位指针。

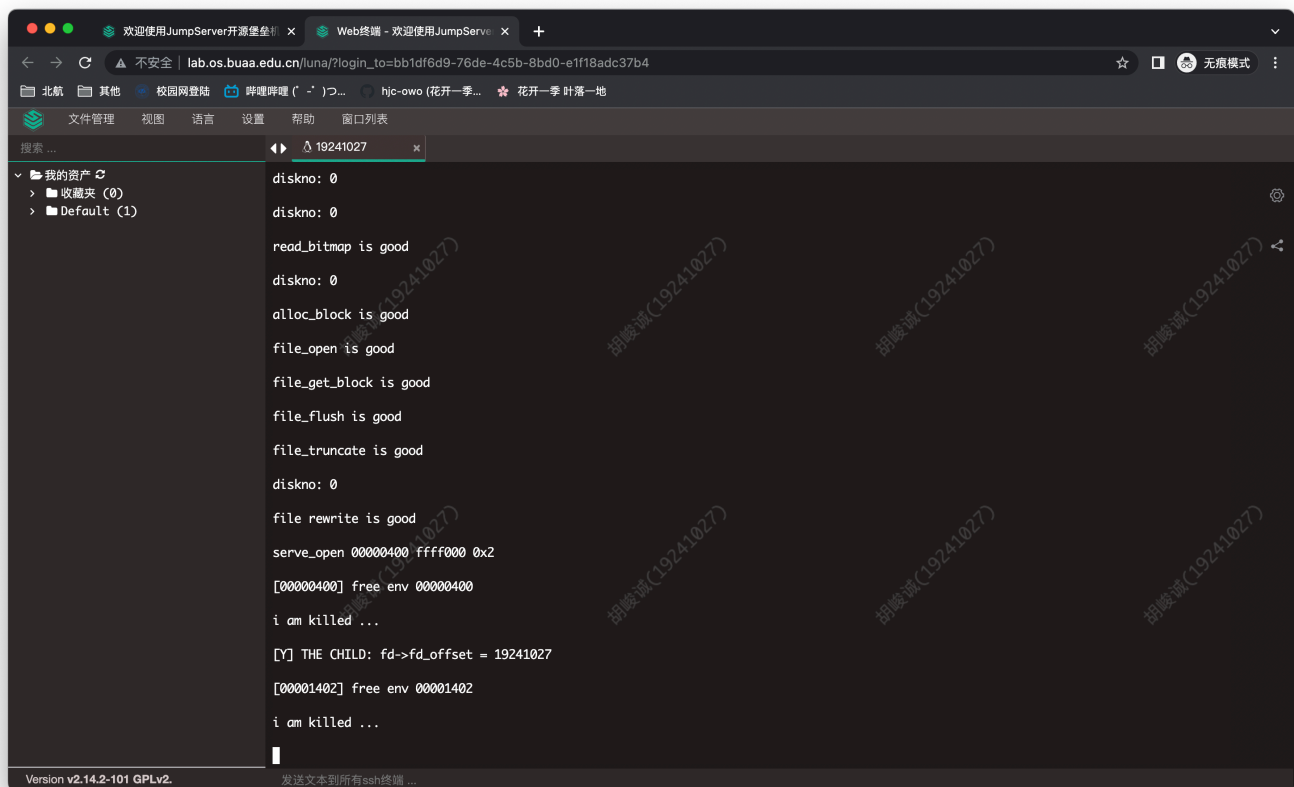
`user/test.c`



```
1 #include "lib.h"
2
3 void umain() {
4     int fdnum = open("/newmotd", O_RDWR);
5     if (fdnum < 0) {
6         user_panic("open failed");
7     }
8     seek(fdnum, 19241027);
9     int pid = fork();
10    if (pid == 0) {
11        struct fd *fd;
12        if (fd_lookup(fdnum, &fd) < 0) {
13            writef(0, "N\n");
14        } else {
15            writef(0, "THE CHILD: fd->fd_offset = %d\n", fd->fd_offset);
16        }
17    }
18 }
```

Version v2.14.2-101 GPLv2. 发送文本到所有ssh终端 ...

运行结果



可见我们 seek 将指针修改为多少，fork 后子进程的 fd\_offset 也为多少。

## Thinking 5.10

请解释 Fd, Filefd, Open 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。

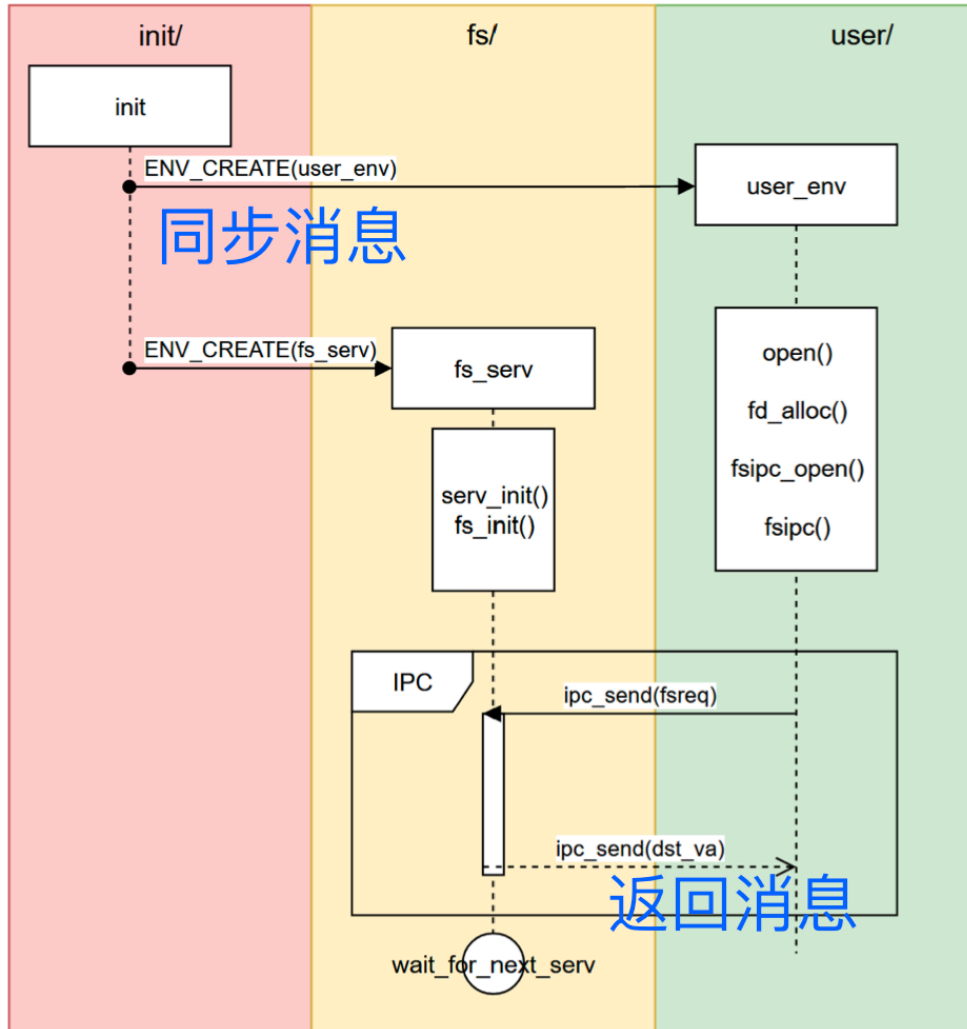
```

1 struct Fd {
2     u_int fd_dev_id; // 指外设id，也就是外设类型。
3     u_int fd_offset; // 读写的当前位置（偏移量），类似“流”的当前位置。
4     u_int fd_omode; // 指文件打开方式，如只读，只写等。
5 };
6
7 struct Filefd {
8     struct Fd f_fd; // 即文件描述符。
9     u_int f_fileid; // 指文件本身的id。
10    struct File f_file; // 指文件本身。
11 };
12
13 struct Open {
14     struct File *o_file; // 指向具体的file。
15     u_int o_fileid; // 即 file 的 id。
16     int o_mode; // 打开方式，指只读，只写等。
17     struct Filefd *o_ff; // 读或写的当前位置，指偏移量。

```

## Thinking 5.11

上图中有多种不同形式的箭头，请结合 UML 时序图的规范，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。



我们的进程通过和 `file_server` 这个进程的通信，来操作文件。

在用户进程中，通过 `user/file.c` 中的函数操作文件系统，在这些用户接口函数中，调用了 `user/fsipc.c` 中的函数，从而通过 `user/fsipc.c` 中的这些函数与文件系统进行了通信。在文件系统进程中，初始化完成后将运行 `serve` 函数，在这个函数中，调用了 `ipc_rcv`，通过返回值的不同（这些返回值定义在 `include/fs.h` 中），在 `switch...case` 语句块中跳转到不同的函数，从而完成通信。

## Thinking 5.12

阅读 `serv.c/serve` 函数的代码，我们注意到函数中包含了一个死循环 `for (;;) {...}`，为什么这段代码不会导致整个内核进入 `panic` 状态？

因为再调用 `ipc_rcv` 的时候这个进程会等待其他进程向他发送请求文件系统调用再继续执行，如果没有接收到请求就会在这里一直等待，因此不会自己执行死循环，导致整个内核进入 `panic` 状态。

## 实验难点

### 1. 设备驱动，即进行设备读写

在我们的实验中，要实现的设备驱动主要有三种：即 console，IDE 和 rtc。在内存中的布局如下表所示。

device	start addr	length
console	0x10000000	0x20
IDE	0x13000000	0x4200
rtc	0x15000000	0x200

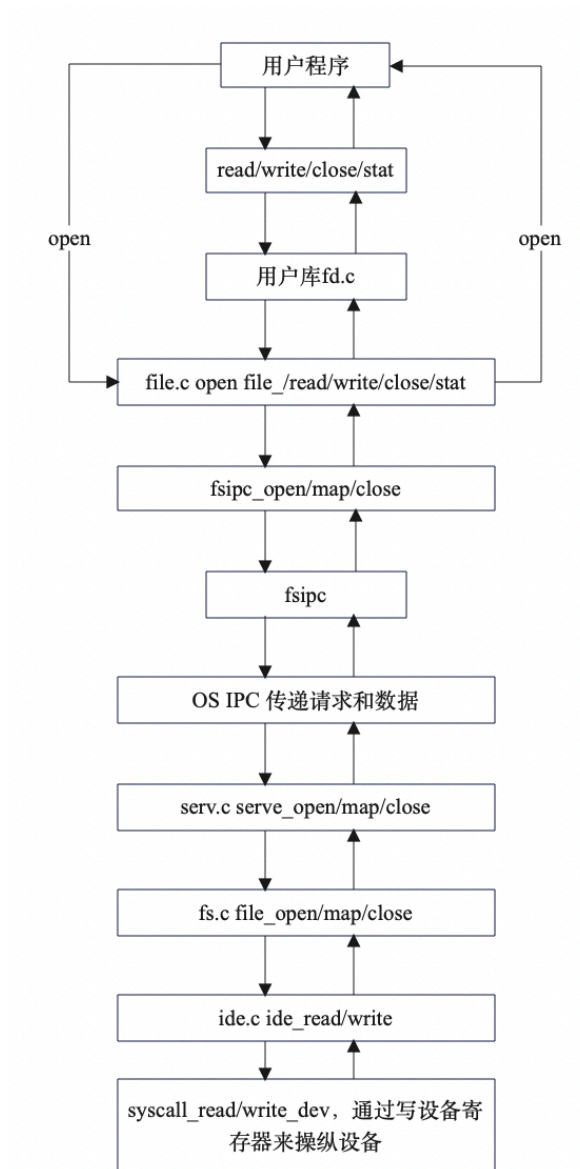
其中：

- console 为控制台终端，即我们在编写程序时用于输入输出的地方。
- IDE 即为磁盘，用于存储文件等。
- rtc 为实时时钟终端，用于获取当前时间等信息。

在用户态对上述外部设备进行读写等操作时，需要用到系统调用，在本次实验中实现的系统调用为 `syscall_write_dev` 与 `syscall_read_dev`。这两个函数实现的是将某段内存中的信息拷贝到外部设备的相应内存区域或将外部设备内存中的信息拷贝到某段内存中。

### 2. 文件系统调用过程





`open` 函数比较特殊，直接由用户调用位于 `file.c` 中的内容而不需要经过 `fd.c` 的抽象

## 体会与感想

好多文件，好多代码。要填的函数只是整个 M0S 冰山一角，更多更多的是需要阅读大量的代码，才能勉强理解整个文件系统的流程。

啥啥磁盘、磁盘块、扇区、文件结构，傻傻分不清楚。

宏定义绕啊绕啊。

这一 lab 的内容综合性较强，将前面几个 lab 的知识体系串联在一起，从 lab2 的内存结构到 lab4 的 IPC，每一步都需要深入理解才能帮助我们顺利完成 lab5 的实验。不要去管文件系统的底层怎么实现的，更多的精力要放在用户调用接口的逻辑上。

不过就剩最后的 lab6 了，M0S 的大厦终将缓缓建成。