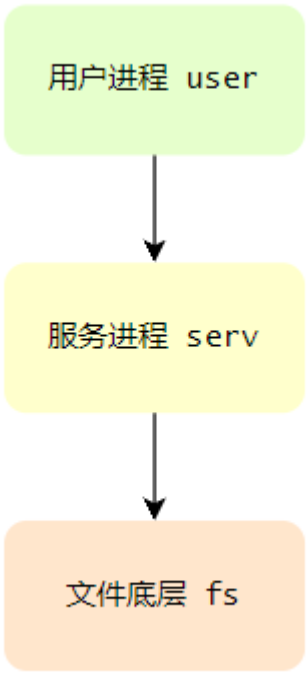
实验目的

- 了解文件系统的基本概念和作用
- 了解普通磁盘的基本结构和读写方式
- 了解实现设备驱动的方法
- 掌握并实现文件系统服务的基本操作
- 了解微内核的基本设计思想和结构

实验难点

其实从 实验目的 就可以看出，文件系统单元涉及的知识并非只是文件系统这么简单。本单元的指导书内容和代码填写量都比较少，但是需要自主阅读的代码量很大，几个主要文件和函数之间的调用关系也比较复杂，因此本章最大的难点或许就是系统、宏观地看待整个文件系统，并基于此梳理文件系统的实现。

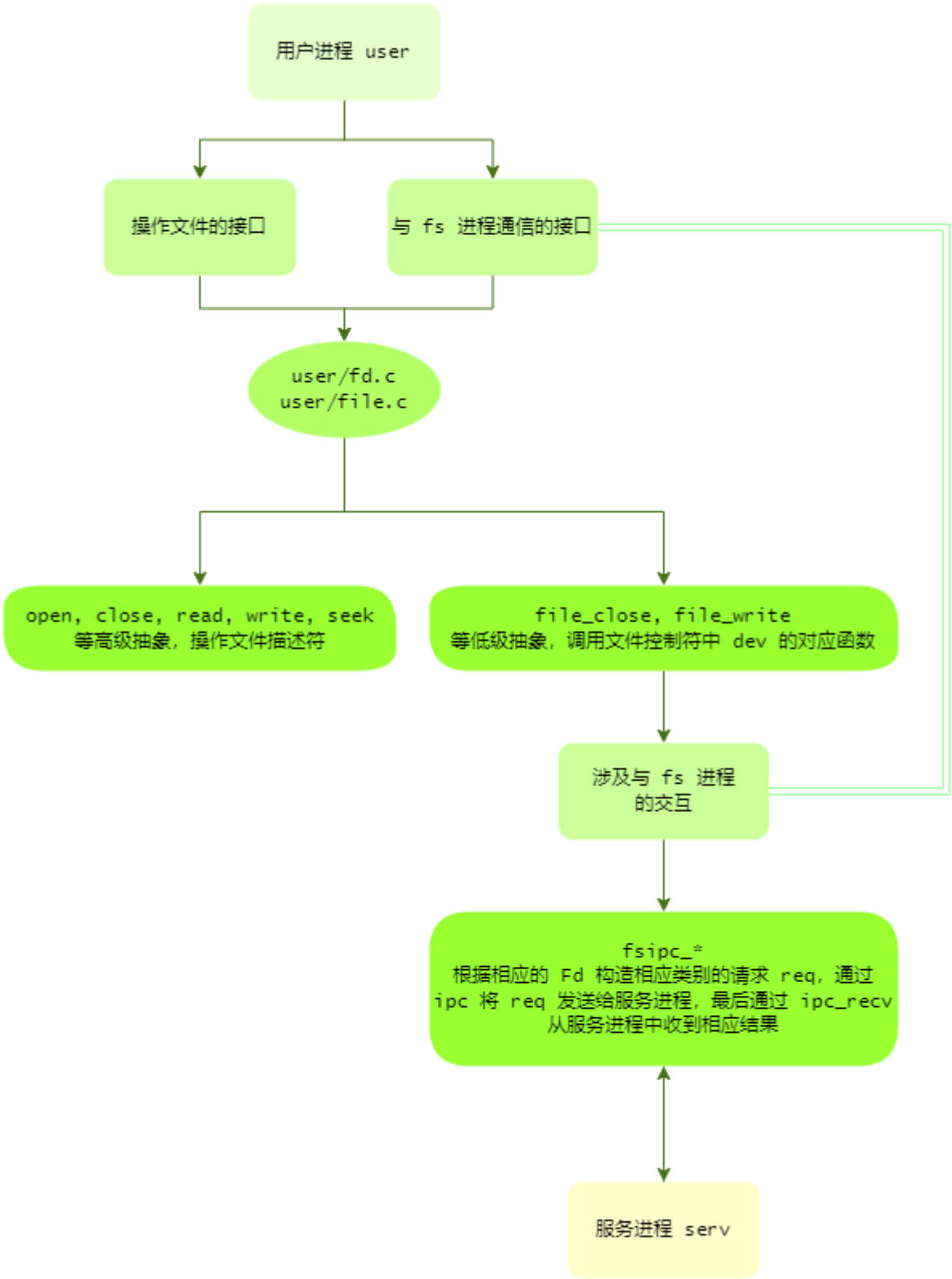


我们的文件系统呈现出如上图所示的明显的抽象封装，在这三个**抽象层次**中，文件的抽象形式不同，用户进程部分最高级。

需要明确的是整个文件系统的服务是由一个另开的用户进程（下文中称为 fs 进程）专门提供的（这里另类地体现了微内核设计的思路），大部分的实现过程都放在 fs 文件夹里，其他进程如果想使用文件系统的服务就需要跟 fs 进程进行通信。在我看来，本单元实验的难点就在于这三个层次之间的协作以及流程回转，繁复的文件让人在读代码时容易搞糊涂，但梳理了个中逻辑后感觉实现过程还是非常清晰的。

用户进程 user

作为顶层的用户进程，实际上这个文件夹中与文件系统相关的文件只需要实现：**使用其文件系统服务所需的接口，与 fs 进程交互所需的函数（进程通信）**。为了完成这两项实现，文件在这个层次被抽象为文件描述符 Fd，操作时只需要操作 Fd 对象。



本层次交互的流程如上图所示。这个时候还不能真的直接和 fs 里的文件进行交互，夹在中间的服务进程 serv 被抽象为了 fs 进程的表象，fs 是底层的本体。

fsipc_* 部分是最令我关注的难点。ipc 将 req 发送给服务进程的本质是将 req 的页面共享给服务进程。与服务进程交互的过程中还存在着一些其他的页面映射，如 fsipc_open 将服务进程中的文件描述符共享映射给用户空间的文件描述符，fsipc_map 将服务进程中相应的 Block 映射给用户进程。此处涉及的 ipc 机制具体流程如下所示。

```
static int fsipc(u_int type, void *fsreq, u_int dstva, u_int *perm)
{
    u_int whom;
    ipc_send(envs[1].env_id, type, (u_int)fsreq, PTE_V | PTE_R);
    return ipc_recv(&whom, dstva, perm);
}
```

概括而言，就是发送方先调用 ipc_send 函数，该函数通过一个死循环来不断向接收方信息；当接收方成功接收到消息时，ipc_send 函数跳出循环并结束，这时发送方再调用 ipc_recv 函数主动放弃 CPU，等待接收返回信息。

> - UserSendToServ
>   - user.fsipc()：根据相应的 Fd 构造相应类别的请求 req，此时页面为 ipc 结构体
>   - ipc_send()
>   - 唤醒被阻塞的 fs 进程
>   - 将 req 的页面共享给 fs 进程：dstva = REQVA 页
>   - ipc_recv()：阻塞用户进程，fs 进程执行相应的文件系统服务
>   - 待写页面为 INDEX2FD(fd) 页
> - ServSendToUser
>   - fs 进程
>   - 保存文件信息于 opentab[i] 中的 o_off 页面
>   - 执行文件系统服务
>   - ipc_send() 唤醒被阻塞的用户进程
>   - 将信息映射给用户进程的待写页面：srcva = o_off 页

在这个层次，文件的抽象形式是文件描述符 Fd。内存中专门开辟了一部分空间 FDTABLE 来存放文件描述符，相当于一个数组，每个 Fd 大小为一页，直接使用 Fdnum 作为数组下标就可以从文件描述符表中获得。文件的具体数据存放在 FILEBASE 部分的内存空间，每个文件数据大小最多为 PDMAP。有了这两个地址空间，可以实现文件描述符和具体文件数据之间的一一映射。

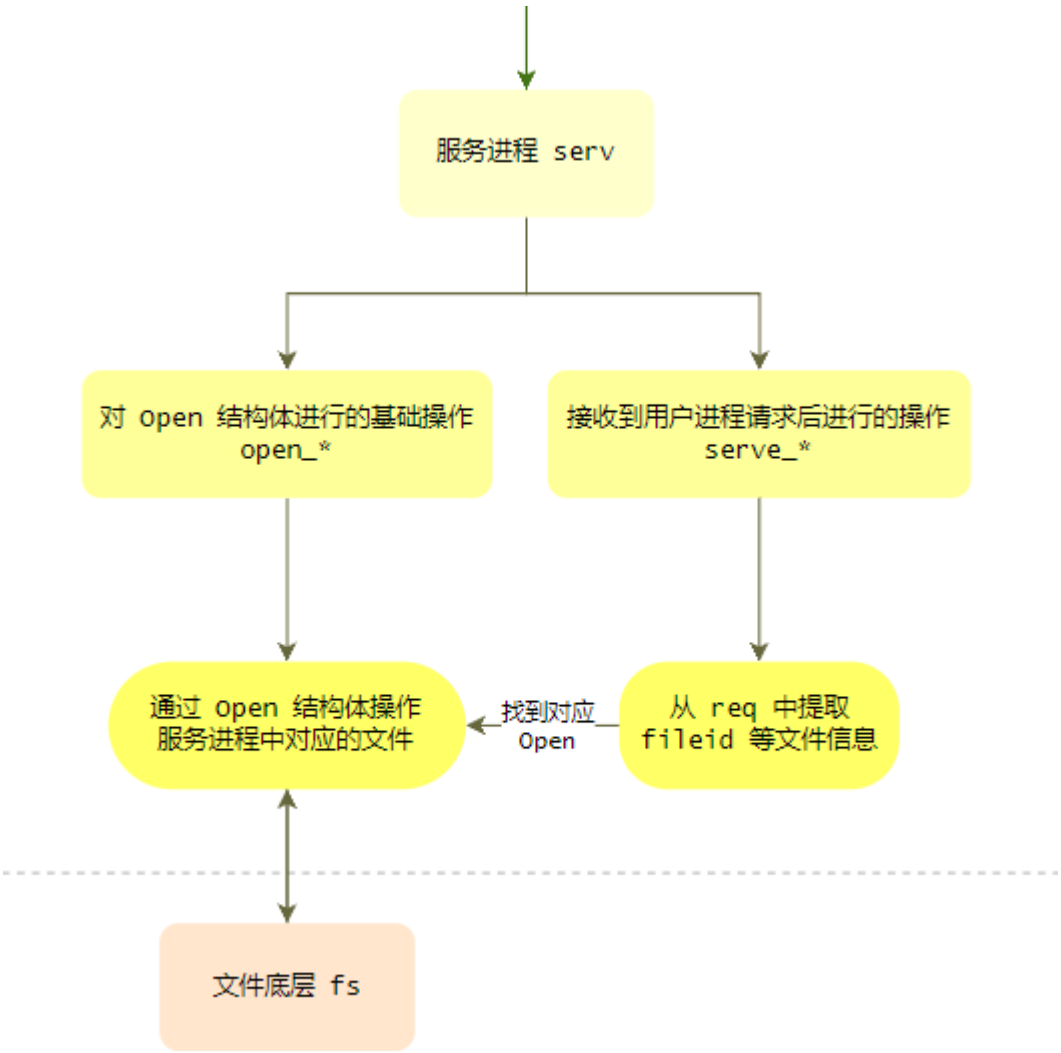
```
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset; // 定位指针
    u_int fd_omode;  // 打开选项，有O_APPEND等
};

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,
};
```

服务进程 serv

文件在这个层次的存在形式比较陌生，是一个名为 **Open** 的结构体，fs/serv.c 中有一个数组 opentable 保存着它，通过 **file_id** 可以在其中查找到对应的 Open 结构体。这里的结构比较难理解，存在着一些嵌套关系。同时，在这个部分引入了**块缓存**的概念，需要注意 Block 和 va 的映射关系：用户进程 - 文件映射区，文件服务进程 - 块缓存。

```
// 主要为文件系统服务进程所用，记录所有进程中被打开的文件
struct Open {
    struct File *o_file;    // 对应的文件控制块指针
    u_int o_fileid;         // 文件id，指示当前打开的文件在 opentable 中的位置
    int o_mode;             // 打开选项
    struct Filefd *o_ff;    // Filefd 结构体
};

// 一般由 Fd 强转而来，存储更丰富的文件信息
struct Filefd {
    struct Fd f_fd;         // 文件描述符
    u_int f_fileid;         // 文件id，指示文件在 opentable 中的位置
    struct File f_file;     // 文件控制块
};

// 主要为用户所用，对应磁盘映射到内存中的数据
struct Fd {
    u_int fd_dev_id;        // 文件对应的设备 id
    u_int fd_offset;        // 定位指针，表征文件指针指向的位置
    u_int fd_omode;         // 打开选项
};
```

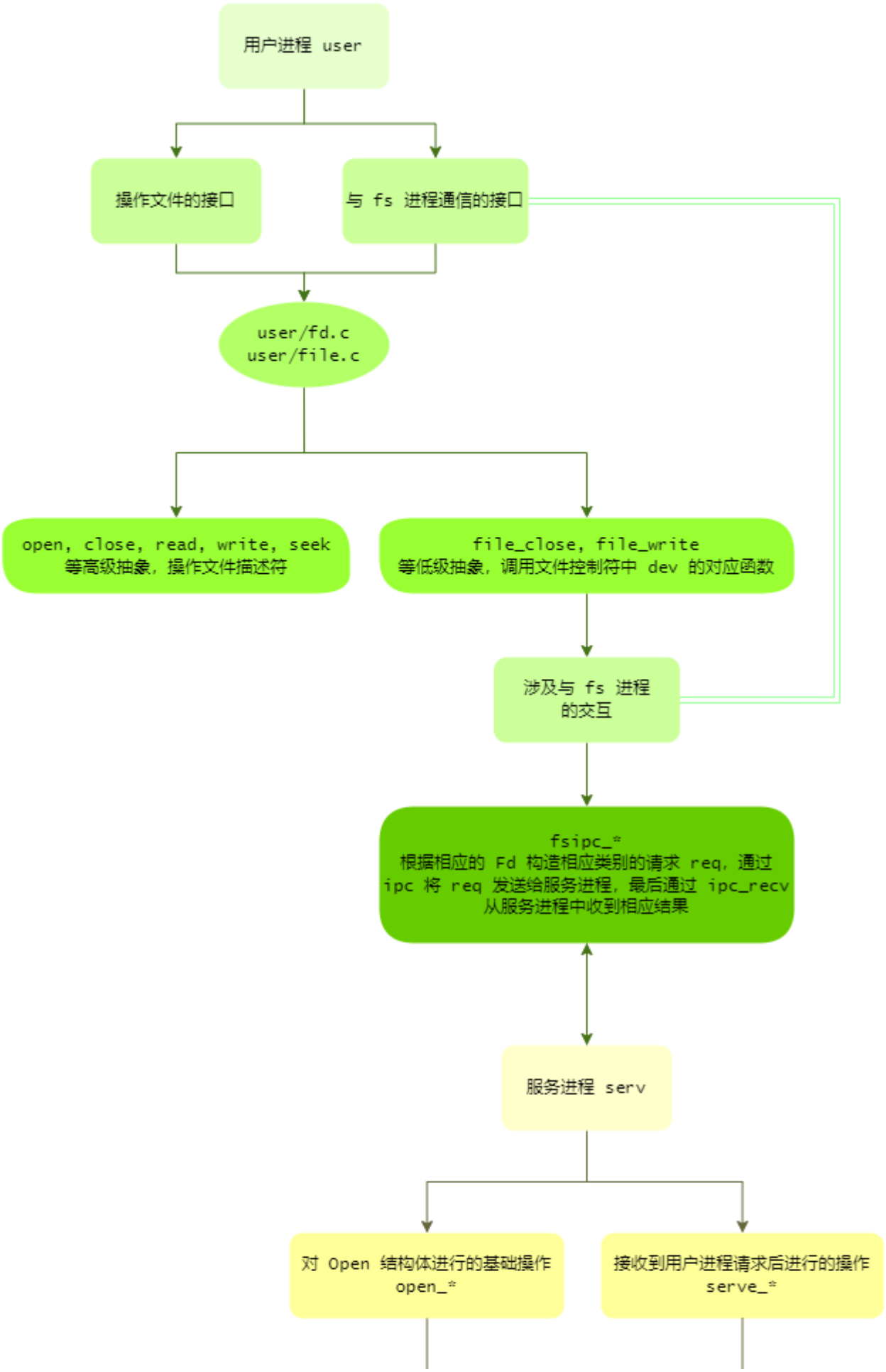
本层次需要实现的功能是**响应来自用户进程的交互请求**，大致可以分为接受请求和操作 Open 结构体两部分。而操作 Open 结构体的过程相当于操作服务进程中对应的具体文件（毕竟保存了文件信息，与 Open 相关的函数可以利用 Open 结构体中保存的文件信息，直接与文件底层进行交互），这时候就可以调用文件底层的函数中来把相应的 Block 从磁盘读入服务进程的内存空间中，或对相应的 Block 进行操作。具体的交互流程如下图所示。

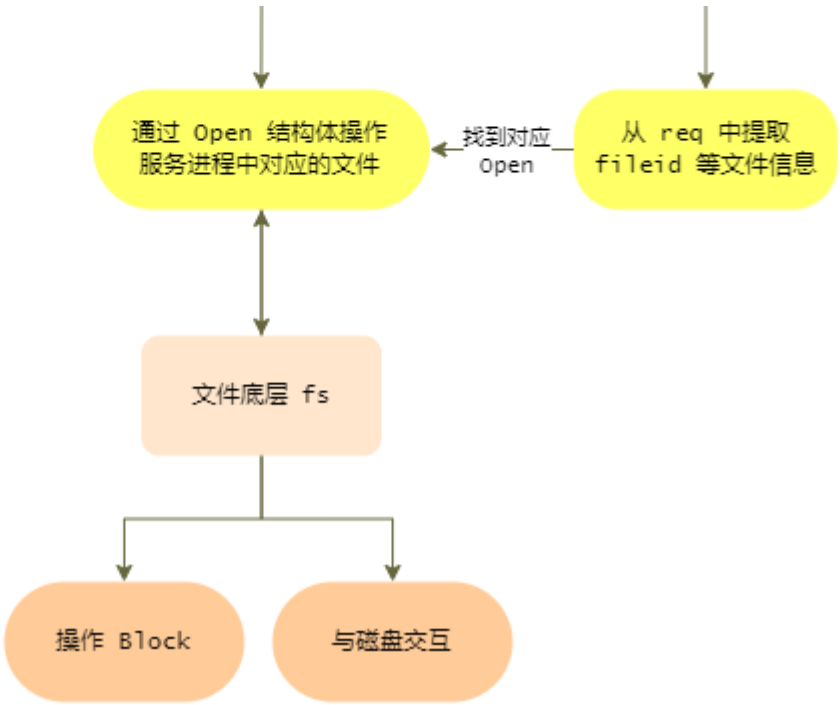


文件底层 fs

服务进程调用此层次的函数以操作文件对应的块，可以知道这部分封装的就是能根据文件修改相应 Block 的函数 file_*，在这里可以完成对 Block 的获取、操作、修改以及和磁盘的交互。文件在这个层次以文件控制块的形式存在，抽象为 File 结构体。

不过需要注意的是对 Block 的修改其实就是对服务进程 serv 内存空间的修改，这里单独拆出来只是为了体现层次化的思想，本质上它跟 serv 是一部分，或者说，**serv + fs = fs 进程。**


用户进程 user
操作文件的接口
与 fs 进程通信的接口
user/fd.c
user/file.c
open, close, read, write, seek
等高级抽象，操作文件描述符
file_close, file_write
等低级抽象，调用文件控制符中 dev 的对应函数
涉及与 fs 进程
的交互
fsipc_*
根据相应的 Fd 构造相应类别的请求 req，通过
ipc 将 req 发送给服务进程，最后通过 ipc_recv
从服务进程中收到相应结果
服务进程 serv
对 Open 结构体进行的基础操作
open_*
接收到用户进程请求后进行的操作
serve_*



至于不同文件的函数之间的协作封装关系图，我则参考了往届博客，并非原创，在报告中就不放出来了，但在实验过程中对我帮助实在很大，不胜感激。

思考题

Thinking 5.1

/proc 文件系统是一个虚拟文件系统，用一系列文件存储了当前内核的运行状态，通过它可以使用一种新的方法在 Linux 内核空间和用户间之间进行通信。在 /proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，但与普通文件不同的是，这些虚拟文件的内容是动态创建的，通过它们可以查看当前进程运行信息或者修改内核运行状态。

Windows 操作系统通过一个可伸缩的系统管理结构 WMI 实现这些功能，它向外暴露一组 API 使得用户可以对操作系统信息以及系统活动进行管理。

/proc 文件系统这样的设计可以方便用户对内核进行管理，由于存在于内存中，它的访问速度也很快。不过这势必会导致该文件系统占据大量内存空间，因此在此处可以进行一些优化改进。

Thinking 5.2

如果通过 kseg0 访问设备，那么缓存块只能再要被更新的数据替换时才被写入内存，引发后续的错误。而对于不同种类的设备，由于串口设备读写频率大于 IDE 磁盘，因此串口设备发生错误的概率也会远大于 IDE 磁盘。

Thinking 5.3

- 链接方式不同

  MOS 中的文件控制块只有一个一级间接指针域，而 Linux 的 inode 支持多级间接指针域，可以存储的文件更大。而且文件控制块是顺次排列在目录文件中，inode 则通过指针与目录项链接，查找操作的系统开销远远小于文件控制块，占用的内存空间也更小。

- 存储文件信息不同

- MOS 中的文件控制块

```
// file control blocks, defined in include/fs.h
struct File {
    u_char f_name[MAXNAMELEN];  // filename
    u_int f_size;               // file size in bytes
    u_int f_type;               // file type
    u_int f_direct[NDIRECT];
    u_int f_indirect;
    struct File *f_dir;
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

  文件名，文件大小，文件类型，文件数据快的直接指针数组，文件数据块的一级间接指针，文件所在目录的指针

- Linux 中的 inode
  - inode 编号
  - 文件的字节数
  - 文件的块数
  - 文件数据块的位置
  - 文件拥有者的 User ID
  - 文件的 Group ID
  - 文件的读、写、执行权限
  - 文件的时间戳
    - ctime：inode 上一次修改时间
    - mtime：文件内容上一次修改时间
    - atime：文件上一次打开时间
  - inode 被链接数

Thinking 5.4

- 一个磁盘块能存储 BY2BLK / BY2FILE = 4096 / 256 = **16** 个文件控制块。
- 一个目录包含 1024 个指向磁盘块的指针，因此最多能有 1024 * 16 = 16384 个文件。
- 一个文件最多可以使用 1024 个磁盘块存储数据，因此文件系统支持的单个文件最大为 1024 * BY2BLK = 4MB。

Thinking 5.5

DISKMAX = 0x40000000 -> 支持的最大磁盘大小为 1GB。

Thinking 5.6

如果将 DISKMAX 改为 0xc0000000，会有一部分的文件数据块存储在内核中，此时处于用户态进程的文件系统无法操作这些内核中的数据块，会报 TOO LOW。

Thinking 5.7

```
// macro in fs/fs.h
#define INDEX2FD(i)     (FDTABLE+(i)*BY2PG)
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)
```

结合 实验难点 部分的分析，第一个宏用于在文件描述符表中利用下标查找需要的文件描述符，第二个则利用下标查找对应的文件信息页面。

```
// file control blocks, defined in include/fs.h
struct File {
    u_char f_name[MAXNAMELEN];  // filename
    u_int f_size;               // file size in bytes
    u_int f_type;               // file type
    u_int f_direct[NDIRECT];
    u_int f_indirect;
    struct File *f_dir;
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

最重要的结构体则应该是文件控制块 File，利用它可以定位文件信息，具体方式则已见于*指导书注释*，对于其中文件信息的利用则见于 实验难点 部分。

> f_name 为文件名称，文件名的最大长度 MAXNAMELEN 值为 128。
>
> f_size 为文件的大小，单位为字节。
>
> f_type 为文件类型，有普通文件 (FTYPE_REG) 和文件夹 (FTYPE_DIR) 两种。
>
> f_direct[NDIRECT] 为文件的直接指针，每个文件控制块设有10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4KB，也就是说，这十个直接指针能够表示最大 40KB 的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。
>
> 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。为了简化计算，我们不使用间接磁盘块的前十个指针。
>
> f_dir 指向文件所属的文件目录。
>
> f_pad 则是为了让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节。

Thinking 5.8

这里我理解成一种形如 OO 中“向下转型”的思想，由于 Filefd 中包含一个 Fd 结构体，它跟被强转的 Fd 指向的是同一个虚拟地址 INDEX2FD(fd)，将 Fd 强转为 Filefd 后依然可以通过 p->f_fd 访问 Fd 中的数据，因此转换不会使得 Fd 指向的数据丢失。

Thinking 5.9

fork 前后的父子进程会共享文件描述符和定位指针。~~(lab5-2-exam乐子)~~

验证程序如下：

```
#include "lib.h"

static char *msg = "test new msg\n\n";
```

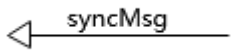
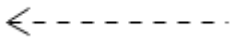
```
static char *diff_msg = "test diffrent msg\n\n";

void umain()
{
        int r, n, id;
        char buf[512];

        if ((r = open("/newmotd", O_RDWR)) < 0) {
            user_panic("open /newmotd: %d", r);
        }
        writef("open is good\n");

        if ((n = read(r, buf, 511)) < 0) {
            user_panic("read /newmotd: %d", r);
        }
        if (strcmp(buf, diff_msg) != 0) {
            user_panic("read returned wrong data");
        }
        writef("read is good\n");

        struct Fd *fdd;
        if ((id = fork()) == 0) {
            if ((n = read(r, buf, 511)) < 0) {
                user_panic("child read /newmotd: %d", r);
            }
            if (strcmp(buf, diff_msg) != 0) {
                user_panic("child read returned wrong data");
            }
            writef("child read is good && child_fd == %d\n", r);
            fd_lookup(r, &fdd);
            writef("child_fd's offset == %d\n", fdd->fd_offset);
        }
        else {
            if((n = read(r, buf, 511)) < 0) {
                user_panic("father read /newmotd: %d", r);
            }
            if (strcmp(buf, diff_msg) != 0) {
                user_panic("father read returned wrong data");
            }
            writef("father read is good && father_fd == %d\n", r);
            fd_lookup(r, &fdd);
            writef("father_fd's offset == %d\n", fdd->fd_offset);
        }
}
```

Thinking 5.10

已在 *实验难点* 的 *服务进程*erv 部分解释。

Thinking 5.11

UML 时序图中有以下几种箭头：

箭头类型	消息类型	消息内涵
	同步消息	消息被发送后，发送者停止活动等待反馈
	异步消息	消息被发送后，发送者继续自己的活动不等待反馈
	返回消息	从过程调用返回
黑实线 + 开三角箭头 + <<create>>	创建消息	发送者发送一个实例化消息后触发的操作，指向被创建对象的表示框
黑实线 + 开三角箭头 + <<destroy>>	销毁消息	将对象销毁并回收其拥有的资源

我们的操作系统使用 IPC 通信机制实现进程通信，本质上是同步消息。具体流程已在 *实验难点* 的 *用户进程user* 部分解释。

Thinking 5.12

因为每次循环都调用 ipc_recv 函数，它被调用后会主动放弃 CPU，使得进程状态被标为 NOT_RUNNABLE，直到 ipc_send 被调用后才会被唤醒，类似一种等待行为，因此不会导致轮询，使得内核进入 panic 状态。

实验体会

操作系统课程即将进入收尾阶段，我在文件系统这一单元也第一次感受到了操作系统内部层次清晰、逻辑分明的调用流程设计，学习的过程很是令人愉悦，从 lab2 暗暗擂来的远古 bug 也让人感受到了各个 lab 之间的环环相扣 😒 ~~（是谁被 96 折磨了两天）~~。

本单元的指导书内容和代码填写量都比较少，如果仅仅是为了完成评测，只阅读注释就绰绰有余。但是文件系统部分需要自主阅读的代码量很大，几个主要文件和函数之间的调用关系也比较复杂，因此本章最大的难点或许就是系统、宏观地看待整个文件系统，阅读代码、理解函数调用关系、梳理文件系统流程，因此本单元令我收获颇丰，写下了字数最长的一篇单元实验报告。

总地来说，这一单元是我目前为止学习体验最好的一个单元，看着这个小小操作系统逐步完善，成就感也不禁油然而生。