操作系统第六次实验报告-Lab5

以下为学生闫思桥(19241091)的$Lab5$实验报告

$Part1$ 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？有什么作用？ Windows 操作系统又是如何实现这些功能的？proc 文件系统这样的设计有什么好处和可以改进的地方？

MyAnswer

- **Linux/Unix 的 /proc 文件系统是什么？有什么作用？**

Linux系统上的/proc目录是一种文件系统，即proc文件系统。与其它常见的文件系统不同的是，/proc是一种伪文件系统（也即虚拟文件系统）。/proc存储的是当前内核运行状态的一系列特殊文件，用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息，甚至可以通过更改其中某些文件来改变内核的运行状态。

内核使用它向外界导出信息，/proc系统只存在内存当中，而不占用外存空间。/proc下面的每个文件都绑定于一个内核函数，用户读取文件时，该函数动态地生成文件的内容。

- **Windows 操作系统又是如何实现这些功能的？**

Windows，分盘，每个驱动器有自己的根目录，形成的是多个树并列的结构。通过 `Win32 API` 的函数调用来实现这些功能。

Linux，只有一个根目录 / ，所有东西都是从这开始。

- **proc 文件系统这样的设计有什么好处和可以改进的地方？**

这样的设计将对内核信息的访问交互抽象成了对文件的访问修改，简化了交互过程。缺点在于其长期驻留内存，占据内存空间。

Thinking 5.2

如果我们通过 kseg0 读写设备，我们对于设备的写入会缓存到 Cache 中。通过 kseg0 访问设备是一种错误的行为，在实际编写代码的时候这么做会引发不可预知的问题。请你思考：这么做这会引起什么问题？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刷新的策略来考虑。

MyAnswer

因为内核被放在kseg0区域，一般通过cache访问；如果对设备的写入缓存到cache中，就会导致以后想访问内核时却错误访问了写入设备的内容。

此外，缓存内存中映射的外部设备内容时，可能外部设备更新后对内存进行了更新，但已经被缓存的部分没有被更新。

这种错误对于磁盘概率较小，串口设备则很容易出现。

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。

MyAnswer

这里放一个链接。本题大部分内容基于这篇博客+我的理解。

Linux中的inode_Chain .的博客-CSDN博客

文件存储在硬盘上，硬盘的最小存储单位叫做“扇区”（Sector）。每个扇区储存512字节。

操作系统读取硬盘的时候，不会一个个扇区的读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个“块”（block）。这种由多个扇区组成的“块”，是文件存取的最小单位。“块”的大小，最常见的是4KB，即连续八个sector组成一个block。

文件数据都储存在“块”中，那么很显然，我们还必须找到一个地方储存文件的“元信息”，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为“索引节点”。

每一个文件都有对应的inode，里面包含了与该文件有关的一些信息。

inode包含文件的元信息，具体来说有以下内容：

- Size 文件的字节数
- Uid 文件拥有者的User ID
- Gid 文件的Group ID
- Access 文件的读、写、执行权限
- 文件的时间戳，共有三个：
  - Change 指inode上一次变动的时间
  - Modify 指文件内容上一次变动的时间
  - Access 指文件上一次打开的时间
- Links 链接数，即有多少文件名指向这个inode
- Inode 文件数据block的位置
- Blocks 块数
- IO Blocks 块大小
- Device 设备号码

而文件控制块的结构体的定义是

```
struct File
{
    /* 除了最后一个成员 其他成员的字节数 MAXNAMELEN + 4 + 4 + NDIRECT*4 + 4 */
    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; // valid only in memory
    u_char f_pad[256 - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
    /* 由此可以得出，一个文件控制块总共有 256 个字节 */
};
```

只含有文件名、文件大小、文件类型（指明它是普通文件还是文件夹/目录）、它指向的磁盘块的直接指针和间接指针、指向文件所属的文件目录的文件控制块指针。最后一个成员没有实际意义，只是为了使得文件控制块的大小占足256字节。

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？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？

MyAnswer

- 一个磁盘块中最多能存储多少个文件 控制块?

结合对文件控制块结构体的分析，可以得出，一个文件控制块的大小是256字节。

`u_char f_pad[256 - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];` 从这个成员可以看出，这个成员就是为了使得文件控制块能占够256字节。

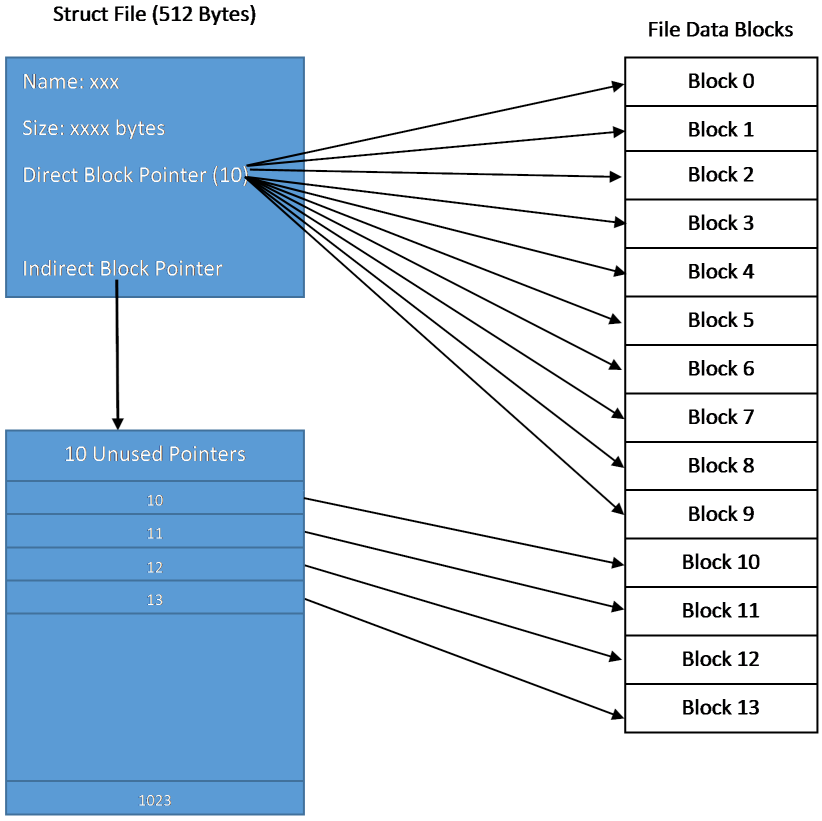
而一个磁盘块的大小是4K，那么，一个磁盘块最多能存储 4K / 256 = 16 个文件控制块。

- 一个目录下最多能有多少个文件?

一个目录下最多能有1024 * 16 = 16K个文件。

在我们的实验中，目录也是文件的一种，也是用文件控制块进行表示的，只不过f_type是 `FTYPE_DIR` 而不是普通文件的 `FTYPE_REG` 。

> 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（**MyNote**：其实就是磁盘块序号）。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。



结合上面的图和文字可以看出，一个文件控制块最多拥有 **一个磁盘块大小的** 指向磁盘块的指针，一个指针类型的数据大小是4字节，那么一个文件控制块最多拥有 BY2BLK/4 = 1024 个指向磁盘块的指针。

> 对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的的文件控制块。

也就是说，一个作为目录的文件控制块最多下属1024个磁盘块，而一个磁盘块最多有16个文件控制块，那么，一个目录最多可以容纳1024 * 16 = 16K个文件（包括子目录）。

- 我们的文件系统支持的单个文件最大为多大?

我们的文件系统支持的单个文件最大是 1024 * BY2BLK = 4M。

还是如上面所引用的那样：对于普通的文件，其指向的磁盘块存储着文件内容。那么上面分析了：一个文件控制块最多拥有 BY2BLK/4 = 1024 个指向磁盘块的指针。这些磁盘块全部用于存放文件内容的话，那么就可以存放 1024 * BY2BLK = 4M 大小的文件内容。这也是单个文件的最大大小。

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?

MyAnswer

最大磁盘大小为 `DISKMAX` = 0x40000000，即1GB。

目前，我们只设计了1024个磁盘块，也就是只占用了4M大小的磁盘空间

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?

MyAnswer

不能，因为缓存磁盘块的时候可能会把内核的内容覆盖掉，导致系统运行异常。

Thinking 5.7

在 lab5 中，fs/fs.h、include/fs.h 等文件中出现了许多结构体和宏定义，写出你认为比较重要或难以理解的部分，并进行解释。

MyAnswer

摘录一些宏的分析。

```
#define BY2BLK      BY2PG       // bytes to block 一个磁盘块的大小是4096字节
#define BIT2BLK     (BY2BLK*8)  // bits to block 一个磁盘块的大小的位(bit)数
#define NBLOCK 1024 // The number of blocks in the disk.
/* 那么可以算出，实验中，一个磁盘的大小其实就是1024 * 4K = 4M */

// File types
#define FTYPE_REG       0   // Regular file
#define FTYPE_DIR       1   // Directory
```

```
// File system super-block (both in-memory and on-disk)

#define FS_MAGIC    0x68286097  // Everyone's favorite OS class

// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN  128
// Maximum size of a complete pathname, including null
#define MAXPATHLEN  1024
// Number of (direct) block pointers in a File descriptor
#define NDIRECT     10
/* Number of block pointers in a Block  */
// 由于间接磁盘块指针f_indirect的存在，一个文件控制块最多可以有一个磁盘块大小的指向磁盘块的指针，一个指针的大小是4个字节（就是磁盘块号）。那么一个文件控制块最多有 BY2BLK/4 = 1024个指向磁盘块的指针。
// 当这个文件控制块的类型f_type是目录文件FTYPE_DIR时，这意味着，一个目录里面最多下辖1024个子文件/子目录
// 当这个文件控制块的类型f_type是普通文件FTYPE_REG时，也就说这1024个它指向的磁盘块存放的全是它的文件内容。因此就不难得出：单个文件最大为 1024*BY2BLK = 4M；也可以理解为，单个文件最多占用1024个磁盘块。
// 也就是这里的 MAXFILESIZE = NINDIRECT * BY2BLK
#define NINDIRECT   (BY2BLK/4)
#define MAXFILESIZE (NINDIRECT*BY2BLK)

#define BY2FILE     256 // bytes to file 文件控制块的字节数
#define FILE2BLK    (BY2BLK/sizeof(struct File))
// files to Block 一个磁盘块最多放置的文件控制块个数，其实就是 4K / 256 = 16 ；sizeof(struct File)也就是 BY2FILE
```

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。

MyAnswer

首先来看两种结构体的定义

```
// file descriptor
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};
// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

那么，因为在结构体Filefd中储存的第一个元素就是struct Fd*，因而对于相匹配的一对struct Fd和struct Filefd，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？请在完成练习5.8和5.9的基础上编写一个程序进行验证。

MyAnswer

先说结论，**fork 前后的父子进程会共享文件描述符和定位指针。**

因为在serve.c 的 `serve_map()` 函数中

```
void serve_map(u_int envid, struct Fsreq_map *rq)
{
    struct Open *pOpen;
    u_int filebno;
    void *blk;
    int r;
    if ((r = open_lookup(envid, rq->req_fileid, &pOpen)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }

    filebno = rq->req_offset / BY2BLK;
    // 找到在文件控制块 pOpen->o_file 下辖的 第 filebno 个磁盘块，将其读入到内存中，并令
*blk 存储这个磁盘块在**虚拟内存中的地址**。
    if ((r = file_get_block(pOpen->o_file, filebno, &blk)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
    ipc_send(envid, 0, (u_int)blk, PTE_V | PTE_R | PTE_LIBRARY);
}
```

在最后一句代码中，函数发送的信息的权限是 `PTE_V | PTE_R | PTE_LIBRARY` ，而PTE_LIBRARY就代表了父子进程是共享某一个页面。此处共享的就是文件描述符所独占的一个页面，自然相应的文件读取定位指针也会被共享。

验证程序这里采用lab5-2-exam里的课上题目的本地测试程序。

> 文件内容如下
>
> ```
> This is a NEW message of the day!
> ```
>
> 代码如下（用下面的内容覆盖 `user/fstest.c` 原有的内容）
>
> ```
> int r, fdnum, n;
> char buf[200];
> fdnum = open("/newmotd", O_RDWR);
> if ((r = fork()) == 0) {
>  n = read(fdnum, buf, 5);
>  writef("[child] buffer is \'%s\'\n", buf);
> } else {
>  n = read(fdnum, buf, 5);
>  writef("[father] buffer is \'%s\'\n", buf);
> }
> ```
>
> 在init.c中启动文件系统服务进程和文件系统测试进程(与lab5课下本地测试一致)
>
> ```
> ENV_CREATE(user_fstest);
> ENV_CREATE(fs_serv);
> ```
>
> 输出如下

```
[father] buffer is 'This '
[child] buffer is 'is a '
```

可见，**目前我们的操作系统在父进程 fork 了子进程后，两个进程会共享文件描述符表，也同时会共用文件描述符的偏移指针。**

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。

MyAnswer

```
// file descriptor
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};
```

Fd结构体用于表示文件描述符，fd_dev_id表示文件所在设备的id，fd_offset表示读或者写文件的时候，距离文件开头的偏移量，fd_omode用于描述文件打开的读写模式。它主要用于在打开文件之后记录文件的状态，以便对文件进行管理/读写，不对应物理实体，只是单纯的内存数据。

```
// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

Filefd结构体是文件描述符 和 文件的组合形式，f_fd记录了文件描述符，f_fileid记录了文件的id，f_file则记录了文件控制块，包含文件的信息以及指向储存文件的磁盘块的指针，对应了磁盘的物理实体，也包含内存数据。

```
struct Open {
    struct File *o_file;    // mapped descriptor for open file
    u_int o_fileid;         // file id
    int o_mode;             // open mode
    struct Filefd *o_ff;    // va of filefd page
};
```

Open结构体在文件系统进程用用于储存文件相关信息，o_file指向了对应文件的文件控制块，o_fileid表示文件id用于在数组opentab中查找对应的Open，o_mode记录文件打开的状态，o_ff指向对应的Filefd结构体。

$Part2$ 实验难点图示

$Part2.1$ 文件系统概述

介绍

为了在磁盘等外部设备上实现我们的文件系统，我们必须为这些外部设备编写驱动程序。实际上，MOS操作系统中已经实现了一个简单的驱动程序，那就是位于driver目录下的串口通信的驱动程序。在这个驱动程序中我们使用了内存映射I/O(MMIO)技术编写驱动。

本次要实现的硬盘驱动程序与已经实现的串口驱动，都采用 MMIO 技术编写驱动，不同之处在于，我们需要驱动的物理设备——IDE 磁盘功能更加复杂，并且本次要编写的驱动程序完全**完全运行在用户空间中**。

本部分我们将首先学习内存映射 I/O，之后了解 IDE 磁盘的结构和操作，最后学习磁盘驱动程序的编写。

内存映射I/O

几乎每一种外设都是通过 **读写设备上的寄存器** 来进行数据通信，外设寄存器也称为**I/O端口**，我们使用I/O端口来访问I/O设备。外设寄存器通常包括控制寄存器、状态寄存器和数据寄存器。这些硬件I/O寄存器被映射到指定的内存空间。

在 Gxemul 中，console设备被映射到 0x10000000，simulated IDE disk被映射到 0x13000000，实时时钟（Real-Time Clock）被映射到0x15000000，等等。更详细的关于 Gxemul 的仿真设备的说明，可以参考Gxemul Experimental Devices。

驱动程序访问的是I/O空间，与一般我们说的内存空间是不同的。外设的I/O空间地址是系统启动后才知道（具体讲，这个辛苦的工作是由BIOS完成后告知操作系统的），通常的体系结构（如x86）并没有为这些已知的外设I/O内存资源的物理地址预定义虚拟地址范围，因此**必须首先将它们映射到内核虚地址空间**，然后驱动程序才能基于虚拟地址及访存指令来实现对I/O设备的操作。

幸运的是，实验中使用的MIPS体系结构并没有复杂的I/O端口的概念，而是统一使用内存映射I/O的模型。MIPS的地址空间中，其在内核地址空间中（kseg0和kseg1段）实现了硬件级别的物理地址和内核虚拟地址的转换机制，其中，对 kseg1 段地址的读写不经过 MMU 映射、不经过高速缓存的特性，正是外部设备驱动所需要的。由于我们是在模拟器上运行操作系统，I/O 设备的物理地址是完全固定的，因此我们可以通过简单地读写某些固定的内核虚拟地址来实现驱动程序的功能。

> 在之前的实验中，我们曾经使用 KADDR 宏把一个物理地址转换为 kseg0 段的内核 虚拟地址，实际上是给物理地址加上 ULIM 的值（即0x80000000）。而在编写设备驱动的时候，我们需要将物理地址转换为 kseg1 段的内核虚拟地址，给物理地址加上 kseg1 的偏移值 (0xA0000000)。

以我们编写完成的串口设备驱动为例，Gxemul 提供的 console 设备的地址为 0x10000000，设备寄存器映射如下表所示

表 6.1: Gxemul Console 内存映射

Offset	Effect
0x00	Read: getchar() (non-blocking; returns 0 if no char is available)
	Write: putchar(ch)
0x10	Read or write: halt()
	(Useful for exiting the emulator.)

这里是说明，在 0x10000000 为基准，在上面两个偏移量对应的地址处设置相应的值，可以达到相应的效果。

现在，我们通过往内存的 (0x10000000+0xA0000000) 地址写入字符，就能在 shell中看到对应的输出。drivers/gxconsole/console.c中的printcharc函数的实现如下所示：

```
 *  * ---------------------------------*
 *  |   device    | start addr | length |
 *  * -----------+------------+--------*
 *  |  console    | 0x10000000 | 0x20   |
 *  |    IDE      | 0x13000000 | 0x4200 |
 *  |    rtc      | 0x15000000 | 0x200  |
 *  * ---------------------------------*
 */
// 理解一下，即：物理设备地址（PHYSADDR）映射到内存虚拟地址的偏移量
#define PHYSADDR_OFFSET     ((signed int)0xA0000000)
#define     DEV_CONS_PUTGETCHAR         0x0000
#define     DEV_CONS_HALT           0x0010
// 理解一下，即：设备（dev）控制台（console）的地址
#define DEV_CONS_ADDRESS        0x10000000
#define DEV_CONS_LENGTH         0x0000000000000020
// kseg1偏移量（PHYSADDR_OFFSET） + 设备（dev）控制台（console）的地址 + 要输出字符串的偏移
#define PUTCHAR_ADDRESS     (PHYSADDR_OFFSET +      \
                DEV_CONS_ADDRESS + DEV_CONS_PUTGETCHAR)
// kseg1偏移量（PHYSADDR_OFFSET） + 设备（dev）控制台（console）的地址 + 要停止（halt）输出的偏移
#define HALT_ADDRESS        (PHYSADDR_OFFSET +      \
                DEV_CONS_ADDRESS + DEV_CONS_HALT)
void printcharc(char ch)
{
    *((volatile unsigned char *) PUTCHAR_ADDRESS) = ch;
}
void halt(void)
{
    *((volatile unsigned char *) HALT_ADDRESS) = 0;
}
void printstr(char *s)
{
    while (*s)
        printcharc(*s++);
}
```

在本次实验中，我们需要编写 IDE 磁盘的驱动完全位于用户空间，用户态进程若是直接读写内核虚拟地址将会由处理器引发一个地址错误（ADEL/S)。所以我们对于设备的读写必须通过系统调用来实现。这里我们引入了 sys_write_dev 和 sys_read_dev 两个系统调用来实现设备的读写操作。这两个系统调用以用户虚拟地址，设备的物理地址和读写的长度（按字节计数）作为参数，在内核空间中完成 I/O 操作。

这里顺便复习一下系统调用函数的接口涉及到的文件

`include/unistd.h` 中，声明系统调用向量号 `#define SYS_*` ，主要是设定该向量号到 `__SYSCALL_BASE` 的偏移量；

`user/lib.h` 中，声明函数 `int syscall_*` ；

`user/syscall_lib.c` 中，实现用户态函数 `int syscall_*` ，用 `syscall_*` 调用 `msyscall(SYS_*, ...)` ；

`lib/syscall.S` 中，需要在 `sys_call_table:` 下以 `.word sys_*` 的形式定义内核态系统调用函数 `sys_*` ；

`lib/syscall_all.c` 中，正式实现内核态系统调用函数 `sys_*` 。

sys_write_dev()

这个函数的含义是：向 被映射过后（+A000 0000）的物理地址 所代表的设备中写入数据。

> **MyNote:** 注释表格中IDE指的就是磁盘，0x4200是Gxemul IDE disk I/O 寄存器相对于 0x13000000 最大的偏移量，具体的偏移量和对应的功能，详见下面的 Gxemul IDE disk I/O 寄存器相对于 0x13000000 的偏移和对应的功能表。

函数具体操作是：从虚拟地址va处（往往是用户进程的低2G虚拟空间）拷贝长度为len的数据到dev+0xa000 0000 处（kseg1区域）。注意这里传入的参数dev是物理地址，因此在使用bcopy的时候需要加上kseg1的基地址 0xa000 0000。

```
/* Overview:
 *  This function is used to write data to device, which is
 *  represented by its mapped physical address.
 *  Remember to check the validity of device address (see Hint below);
 *
 * Pre-Condition:
 *      'va' is the starting address of source data, 'len' is the
 *      length of data (in bytes), 'dev' is the physical address of
 *      the device
 *
 * Post-Condition:
 *      copy data from 'va' to 'dev' with length 'len'
 *      Return 0 on success.
 *  Return -E_INVAL on address error.
 *
 * Hint: Use ummapped segment in kernel address space to perform MMIO.
 *   Physical device address:
 *  * ---------------------------------*
 *  |   device    | start addr | length |
 *  * -----------+------------+--------*
 *  |  console    | 0x10000000 | 0x20   |
 *  |    IDE      | 0x13000000 | 0x4200 |
 *  |    rtc      | 0x15000000 | 0x200  |
 *  * ---------------------------------*
 */
int sys_write_dev(int sysno, u_int va, u_int dev, u_int len)
{
    if (dev >= 0x10000000 && dev + len <= 0x10000020 ||
        dev >= 0x13000000 && dev + len <= 0x13004200 ||
        dev >= 0x15000000 && dev + len <= 0x15000200)
    {
        bcopy(va, 0xa0000000 + dev, len);
        return 0;
    }
    return -E_INVAL;
}
```

sys_read_dev()

这个函数的含义是：从 被映射过后（+A000 0000）的物理地址 所代表的设备中读出数据写入内存。

函数具体操作是：从 dev+0xa000 0000 处（kseg1区域）拷贝长度为len的数据到内核虚拟地址va处（往往是用户进程的低2G虚拟空间）。注意这里传入的参数dev是物理地址，因此在使用bcopy的时候需要加上kseg1的基地址 0xa000 0000。

```
/* Overview:
 *  This function is used to read data from device, which is
 *  represented by its mapped physical address.
 *  Remember to check the validity of device address (same as sys_write_dev)
 *
 * Pre-Condition:
```

```
 *      'va' is the starting address of data buffer, 'len' is the
 *      length of data (in bytes), 'dev' is the physical address of
 *      the device
 *
 * Post-Condition:
 *      copy data from 'dev' to 'va' with length 'len'
 *      Return 0 on success, < 0 on error
 *
 * Hint: Use ummapped segment in kernel address space to perform MMIO.
 */
int sys_read_dev(int sysno, u_int va, u_int dev, u_int len)
{
    if (dev >= 0x10000000 && dev + len <= 0x10000020 ||
        dev >= 0x13000000 && dev + len <= 0x13004200 ||
        dev >= 0x15000000 && dev + len <= 0x15000200)
    {
        bcopy(0xa0000000 + dev, va, len);
        return 0;
    }
    return -E_INVAL;
}
```

需要强调的是，上面两个系统调用只是在设备对应的I/O 寄存器中写入/读出相应的值，从而触发设备执行相应的功能、检查设备执行的结果等，并不是直接读写设备的内容。

而在其中，I/O 寄存器通过相对于 `0x13000000` 的偏移为 `0x4000-0x4200` 的读写缓冲区(data buffer)，实现以扇区大小为单位(512 bytes) 的磁盘设备读写。

在执行读取操作后, 设备相当于把一个扇区的数据放在了内核态的 `0xA000 0000 + ( 0x4000~0x41ff )` , 然后进程从这个缓冲区里取数据。写入操作则是倒过来。

补充：关于时钟中断和实时时钟RTC(Real-Time Clock)

include/cp0regdef.h

```
CR(Cause Register)寄存器的8-15位为中断号
#define STATUSF_IP2 0x400       # 控制台中断 2号中断
#define STATUSF_IP4 0x1000      # 时钟中断   4号中断
#define STATUS_CU0 0x10000000   # CU0，表示SR寄存器第28位 置1，允许用户态下使用CP0寄存器
#define STATUS_KUC 0x2
```

include/kclock.h

```
/* See COPYRIGHT for copyright information. */
#ifndef _KCLOCK_H_
#define _KCLOCK_H_
#define IO_RTC      0xb5000100      /* RTC port */
// RTC就是实时时钟 Real-Time Clock
#ifndef __ASSEMBLER__
void kclock_init(void);
#endif /* !__ASSEMBLER__ */
#endif
```

那么这里定义的宏 `IO_RTC` ，其实就是 kseg1的基地址 `0xA0000 0000` + 时钟的物理基地址 `0x1500 0000` + 时钟设置时钟中断频率的I/O寄存器对应的偏移量 `0x0100` 。

这个头文件定义了时钟初始化函数 `kclock_init()` ，而 `kclock_init` 调用了 `set_timer()` 函数。

lib/kclock.c

```
/* See COPYRIGHT for copyright information. */
/* The Run Time Clock and other NVRAM access functions that go with it. */
/* The run time clock is hard-wired to IRQ4. */
#include <kclock.h>
extern void set_timer();
void kclock_init(void)
{
    set_timer();
}
```

lib/env.c/env_alloc() 初始化进程的时候要修改SR寄存器的IM(Interrupt Mask)，确保相应的中断可以被响应

```
e->env_tf.cp0_status = 0x1000100c;// 只有时钟中断时
```

`0x0001 0000 0000 0000 0001 0000 0000 1100` 第28位、第12位、第2、3位（lab4）

```
e->env_tf.cp0_status = 0x1000140c;// 时钟中断+控制台中断时
```

`0x0001 0000 0000 0000 0001 0100 0000 1100` 第28位、第12位、第10位、第2、3位（去年lab5-1-extra）

lib/kcons_asm.S/set_timer

这个函数是init/init.c中的函数mips_init()中调用的最后一个函数，但是它本身又调用了很多函数

复习前面提到的

第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。

第12bit 设置为1，表示 4 号中断可以被响应。（从第8位到第15位表示0-7号中断）

第0bit `IEc` 为1, 意味着CPU会响应中断； `IEc` 为0, 意味着CPU不会响应中断.

```
.macro  setup_c0_status set clr
    .set    push
    mfc0    t0, CP0_STATUS      # 将CP0_STATUS的内容存入 t0
    or      t0, \set|\clr       # 将t0（SR寄存器）的第0、12、28位置1  IEc 4号中断 CU0
    # (0bit)允许中断 (12bit) 4 号中断可以被响应 (28bit)允许在用户模式下使用 CP0 寄存器
    xor     t0, \clr            # 将t0和0进行异或，
    mtc0    t0, CP0_STATUS
    .set    pop
.endm

    .text
LEAF(set_timer)

    li t0, 0xc8                 # 0xc8表示1秒钟中断200次
    # 0xb5000000 是模拟器(gxemul) 映射实时钟的位置。偏移量为0x100 表示来设置实时钟中断的频率，0xc8 则表示1 秒钟中断200次，如果写入0，表示关闭实时钟。
    sb t0, 0xb5000100
    sw  sp, KERNEL_SP           # 初始化内核栈顶 KERNEL_SP 为 内核栈sp
setup_c0_status STATUS_CU0|0x1001 0     # 0x10000000|0x1001 = 0x10001001
# 那么如果这里是控制台中断，那就是 |0x401        0x10001401
    jr ra
```

```
    nop
END(set_timer)
```

lib/genex.S/handle_int 处理中断（interrupt）

主要效果为：查Cause寄存器，通过掩码操作，获知当前的中断线。正如前面对Cause寄存器介绍的那样，在已经确定了 `excEode` 位为0，代表中断的前提下，接下来在hand_int函数中要做的就是查Cause寄存器的IP部分（8-15位）若为4号中断（时钟中断），则直接调用 `sched_yield` 。

由于我们的实验中只有 `STATUSF_IP4(0x1000)` 这一个 **8-15位上的某一位为1，其余为0**（第12位为1） 的宏，所以hand_int函数也只会处理一种中断——4号中断/时钟中断。

在2021年的 `lab5-1-extra` 中，新增了 `STATUSF_IP2(0x400)` 这个宏（第10位为1），他代表2号中断，由控制台触发的中断。

```
    .set noreorder
    .align 5
NESTED(handle_int, TF_SIZE, sp)
    SAVE_ALL    # 现在异常处理栈中已经有了异常发生时的上下文环境
    .set at
    mfc0    t0, CP0_CAUSE           # 将CP0_CAUSE存入t0
    mfc0    t2, CP0_STATUS          # 将CP0_STATUS存入t2
    and     t0, t2                  # 将t0 t2与运算，结果存入t0
    # 这里记录一下：SR寄存器和Cause寄存器的值进行与运算，主要是为了看第8-15位，
    # 两个寄存器的这些位必须同时为1，这些位对应的中断才会触发
    andi    t1, t0, STATUSF_IP4     # t0和0x1000与运算，即取t0的第12位。存入t1
    # 也就是说，这里第三个参数换成其他的宏，只要是8-15位上的某一位为1，其余为0，就是另一种中断
    bnez    t1, timer_irq           # t1不为0，即为1，说明4号中断触发，进入timer_irq:
    nop
    # 同理，控制台中断也可以这样（下面代码是自己加上的）
    //andi  t1, t0, STATUSF_IP2     # t0和0x1000与运算，即取t0的第10位。存入t1
    //bnez  t1, cons_irq            # t1不为0，即为1，说明2号中断触发，进入cons_irq:
    //nop
END(handle_int)

timer_irq:
    sb zero, 0xb5000110             # 写 0xb5000110 地址响应时钟中断
                                    # 读取或写入：确认一个计时器中断
1:  j   sched_yield
    nop
    /*li t1, 0xff
    lw    t0, delay
    addu  t0, 1
    sw  t0, delay
    beq t0,t1,1f
    nop*/
    j   ret_from_exception
    nop

cons_irq:
1:  jal handle_cons_ir
    nop
    j   ret_from_exception
    nop
```

lib/Makefile

```
all: env.o print.o printf.o sched.o env_asm.o kclock.o traps.o genex.o
kclock_asm.o syscall.o syscall_all.o getc.o kernel_elfloader.o handle_cons_ir.o
kcons.o kcons_asm.o  # 这三个是新加的
```

IDE磁盘

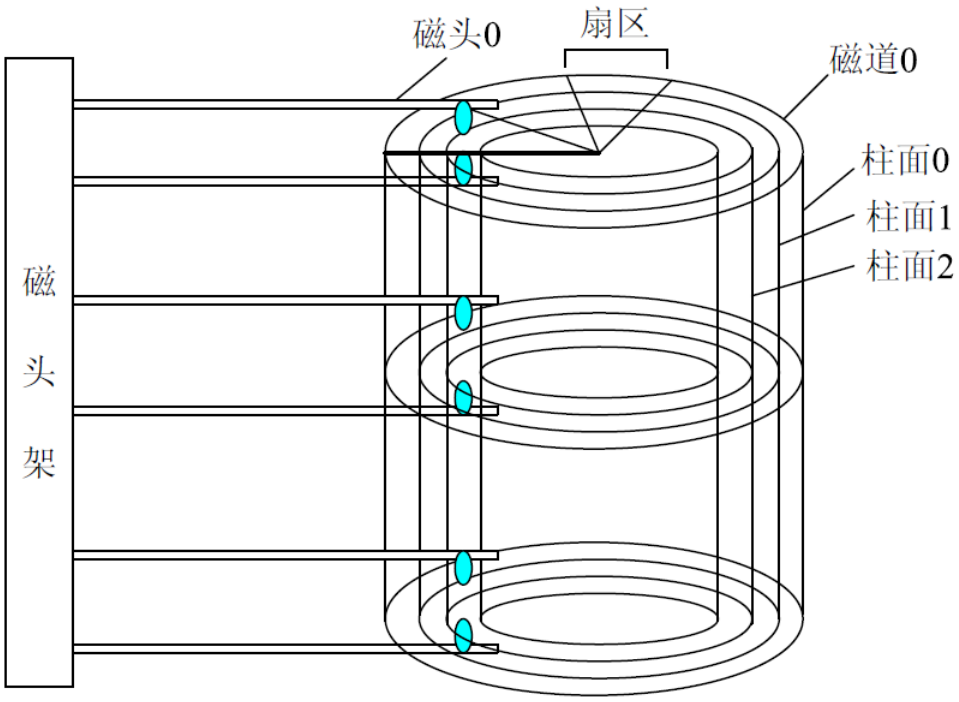
在我们的操作系统实验中，我们使用的 Gxemul 模拟器提供的“磁盘”是一个IDE仿真设备，我们需要此基础上实现我们的文件系统。

磁盘的物理结构

磁盘相关的几个基本概念：

1. 扇区(sector)：磁盘盘片被划分成很多扇形的区域，叫做扇区。扇区是磁盘执行读写操作的单位，一般是512 字节。扇区的大小是一个磁盘的硬件属性。
2. 磁道(track): 盘片上以盘片中心为圆心，不同半径的同心圆。
3. 柱面(cylinder)：硬盘中，不同盘片相同半径的磁道所组成的圆柱。
4. 磁头(head)：每个磁盘有两个面，每个面都有一个磁头。当对磁盘进行读写操作时，磁头在盘片上快速移动。

典型的磁盘的基本结构如图所示：



IDE 磁盘操作

扇区 (Sector) 是磁盘读写的基本单位，Gxemul 也提供了对扇区进行操作的基本方法。对于 Gxemul 提供的模拟 IDE 磁盘（Simulated IDE disk），我们可以把它当作真实的磁盘去读写数据，通过读写特定位置实现数据的读写以及查看读写是否成功。

Gxemul 提供的 Simulated IDE disk 的地址是 0x13000000，I/O 寄存器相对于 0x13000000 的偏移和对应的功能如下图所示。

表 6.2: Gxemul IDE disk I/O 寄存器映射

Offset	Effect
0x0000	Write: Set the offset (in bytes) from the beginning of the disk image. This offset will be used for the next read/write operation.
0x0008	Write: Set the high 32 bits of the offset (in bytes). (*)
0x0010	Write: Select the IDE ID to be used in the next read/write operation.
0x0020	Write: Start a read or write operation. (Writing 0 means a Read operation, a 1 means a Write operation.)
0x0030	Read: Get status of the last operation. (Status 0 means failure, non-zero means success.)
0x4000-0x41ff	Read/Write: 512 bytes data buffer.

驱动程序编写

通过对printcharc 函数的实现的分析，我们已经掌握了 I/O 操作的基本方法，那么，读写 IDE 磁盘的相关代码也就不难理解了。我们以从硬盘上读取一些扇区为例，先了解一下内核态的驱动是如何编写的：

内核态磁盘驱动 read_sector

```
# read sector at specified offset from the beginning of the disk imgae
LEAF(read_sector)
    sw  a0, 0xB3000010  # select the IDE id
    sw  a1, 0xB3000000  # offset
    li  t0, 0
    sb  t0, 0xB3000020  # start read
    lw  v0, 0xB3000030  # get the status of the last operation
                        # 这里的上一个就是指刚刚开始的读/写
    nop
    jr  ra
    nop
END(read_sector)
```

当需要从磁盘的指定位置读取一个 sector 时，我们需要调用 read_sector 函数来将磁盘中对应 sector 的数据读到设备缓冲区中。注意，**所有的地址操作都需要将物理地址转换成虚拟地址**。此处设备基地址对应的 kseg1 的内核虚拟地址是 0xB3000000。

首先，设置 IDE disk 的 ID，read_sector 函数的声明 为 extern int read_sector(int diskno, int offset)，从中可以看出，diskno 是第一个参数，对应的就是 $a0 寄存器的值，因此，将其写入到 0xB3000010 处，这样就表示我们将使用编号为 $a0 的磁盘。在本实验中，只使用了一块 simulated IDE disk，因此，这个值应该为 0。

接下来，将相对于磁盘起始位置的 offset 写入到 0xB3000000 位置，表示在距离磁盘起始处 offset 的位置开始进行磁盘操作。然后，根据 Gxemul 的 data sheet(表6.2)，向内存 0xB3000020 处写入 0 来开始读磁盘（如果是写磁盘，则写入 1）。

最后，将磁盘操作的状态码放入 $v0 中，作为结果返回。通过判断 read_sector 函数的返回值，就可以知道读取磁盘的操作是否成功。如果成功，**将这个 sector 的数据 (512 bytes) 从设备缓冲区 (offset 0x4000-0x41ff) 中拷贝到目的位置。**至此，就完成了对磁盘的读操作。

写磁盘的操作与读磁盘的一个区别在于：**写磁盘需要先将要写入对应 sector 的 512 bytes 的数据放入设备缓冲中**，然后向地址 0xB3000020 处写入 1 来启动操作，并从 0xB3000030 处获取写磁盘操作的返回值。

相应地，**用户态磁盘驱动**使用系统调用代替直接对内存空间的读写，从而完成寄存器配置和数据拷贝等功能。

用户态磁盘驱动 ide_read()

从序号为diskno的磁盘读出数据复制到虚拟内存空间。从序号为secno的扇区开始，总计读出nsecs个扇区的数据。数据最终复制/加载到了 dst 代表的地址处。

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
{
    // 0x200: the size of a sector: 512 bytes.
    int offset_begin = secno * 0x200;
    int offset_end = offset_begin + nsecs * 0x200;
    int offset = 0;

    u_int zero = 0;
    u_int cur_offset = 0;

    while (offset_begin + offset < offset_end) {
        // error occurred, then panic.
        // select the IDE id to read
        /* 再次复习C语言，这里就是把diskno的值写入了0xb3000010
          这里是使用&引用的方式，把 diskno 的地址传进去了，
          这样在sys_write_dev()函数里调用bcopy的时候，就会解引用，把 diskno 的值写入
0xb3000010 */
        if (syscall_write_dev((u_int)&diskno, 0x13000010, 4) < 0)
            user_panic("ide_read panic");
        // offset
        cur_offset = offset_begin + offset;
        /* 把距离磁盘镜像基地址的偏移量 cur_offset 的值写入到 0x13000000 */
        if (syscall_write_dev((u_int)&cur_offset, 0x13000000, 4) < 0)
            user_panic("ide_read panic");
        // start read
        /* 向 0x13000020 写入0来开启读操作 */
        if (syscall_write_dev((u_int)&zero, 0x13000020, 4) < 0)
            user_panic("ide_read panic");
        // get status of last operation(read)
        u_int succ = 0;
        /* 从0x13000030中读出本次读的结果状态，并且写入到 succ */
        if (syscall_read_dev((u_int)&succ, 0x13000030, 4) < 0)
            user_panic("ide_read panic");
        if (!succ)
            user_panic("ide_read panic");
        // IDE读操作结束以后，数据已经被加载到了缓存区，所以最终要从缓存区读出数据到目标地址
        if (syscall_read_dev((u_int)(dst + offset), 0x13004000, 0x200) < 0)
            user_panic("ide_read panic");
        offset += 0x200;// 读取下一个扇区
    }
}
```

用户态磁盘驱动 ide_write()

向序号为diskno的磁盘写入数据。从序号为secno的扇区开始，总计写入nsecs个扇区的数据。数据来源是 src 代表的地址处。

```
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
{
    int offset_begin = secno * 0x200;
    int offset_end = offset_begin + nsecs * 0x200;
    int offset = 0;
```

```
    u_int one = 1;
    u_int cur_offset = 0;

    // DO NOT DELETE WRITEF !!!
    // writef("diskno: %d\n", diskno);

    while (offset_begin + offset < offset_end)
    {
        // copy data from source array to disk buffer.
        // if error occur, then panic.
        /* 写磁盘需要先将要写入对应 sector 的 512 bytes 的数据放入设备缓冲 0x13004000 中
*/
        if (syscall_write_dev((u_int)(src + offset), 0x13004000, 0x200) < 0)
            user_panic("ide_write panic");
        // select the IDE id to write
        if (syscall_write_dev((u_int)&diskno, 0x13000010, 4) < 0)
            user_panic("ide_write panic");
        // offset
        cur_offset = offset_begin + offset;
        if (syscall_write_dev((u_int)&cur_offset, 0x13000000, 4) < 0)
            user_panic("ide_write panic");
        // start write
        /* 向 0x13000020 写入0来开启读操作 */
        if (syscall_write_dev((u_int)&one, 0x13000020, 4) < 0)
            user_panic("ide_write panic");
        // get status of last operation(write)
        u_int succ = 0;
        /* 从0x13000030中读出本次写的结果状态，并且写入到 succ */
        if (syscall_read_dev((u_int)&succ, 0x13000030, 4) < 0)
            user_panic("ide_write panic");
        if (!succ)
            user_panic("ide_write panic");
        offset += 0x200;// 写入下一个扇区
    }
}
```

Part2.2 文件系统结构

实现了IDE磁盘的驱动，我们就有了**在磁盘上实现文件系统**的基础。接下来我们设计整个文件系统的结构，并在磁盘和操作系统中分别实现对应的结构。

> Note：Unix/Linux 操作系统一般将磁盘分成两个区域：inode 区域和 data 区域。inode 区域用来保存文件的状态属性，以及指向数据块的指针。data 区域用来存放文件的内容和目录的元信息 (包含的文件)。MOS 操作系统的文件系统采用了类似的设计，但需要注意与 Unix/Linux 操作系统的区别。

首先明确一点：

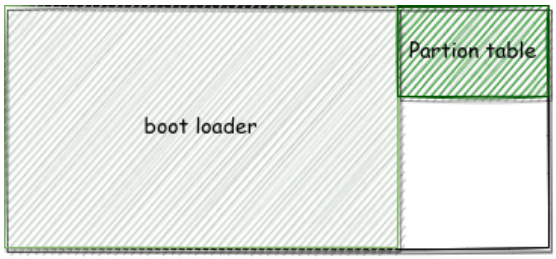
本实验中，所谓的“指向磁盘块的指针”，其实就是这个磁盘块在磁盘块数组 `Struct Block disk[NBLOCK]` 中的下标，也就是在整个磁盘中的磁盘块号码。并不是真正意义上的地址。

先介绍一些新的宏、全局变量、枚举和结构体的含义

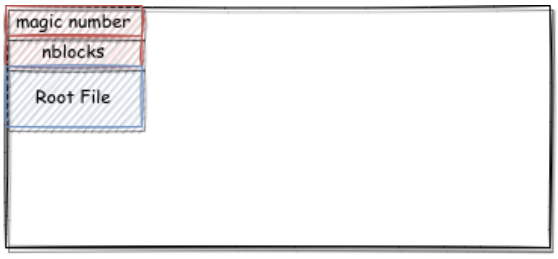
```
/* in fsformat.c  */
#define BY2BLK      BY2PG       // bytes to block 一个磁盘块的大小是4096字节    4K
#define BIT2BLK     (BY2BLK*8)  // bits to block 一个磁盘块的大小的位(bit)数
#define NBLOCK 1024 // The number of blocks in the disk.
/* 那么可以算出，实验中，我们占用的磁盘的大小磁盘的大小其实就是1024 * 4K = 4M */
/* 但是要注意，磁盘的大小实际上是1GB，这点在fs.h中的宏会体现 */
```

```
uint32_t nbitblock; // the number of bitmap blocks.
uint32_t nextbno;   // next availiable block.

// File types
#define FTYPE_REG       0   // Regular file
#define FTYPE_DIR       1   // Directory
// File system super-block (both in-memory and on-disk)

#define FS_MAGIC    0x68286097  // Everyone's favorite OS class
enum {
    BLOCK_FREE  = 0,     // 空闲磁盘块
    BLOCK_BOOT  = 1,     // 用作启动整个磁盘的启动磁盘块，也会存放分区表
    BLOCK_BMAP  = 2,     // 存放位图的磁盘块，标识磁盘中的每个磁盘块的使用情况。在我们的磁盘里，一共只有1k = 1024个磁盘块，而一个磁盘块有4K*8 bit，远大于1k，所以一个磁盘块足够了
    BLOCK_SUPER = 3,     // 超级磁盘块
    BLOCK_DATA  = 4,     // 数据磁盘块
    BLOCK_FILE  = 5,     // 文件磁盘块，由于磁盘块是4K，文件控制块是256B，则可以存放16个文件控制块
    BLOCK_INDEX = 6,     // 指针（磁盘序号）记录磁盘块，用于给某个文件控制块 struct file 记录其下辖的磁盘块的间接指针（即磁盘序号）
};
struct Block {
    uint8_t data[BY2BLK];   // 一个磁盘块的数据是 BY2BLK=4K 个字节
    uint32_t type;
} disk[NBLOCK]; // 定义了磁盘块。disk作为磁盘块 Block 的数组，充当了“磁盘”这一角色
```

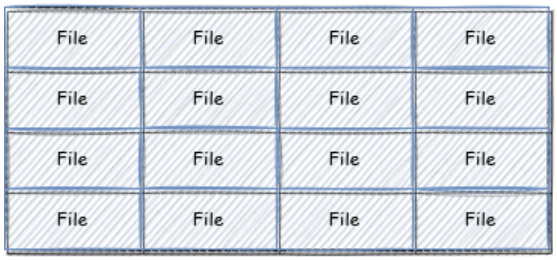
这里放一张强生大佬的图，便于总体理解：



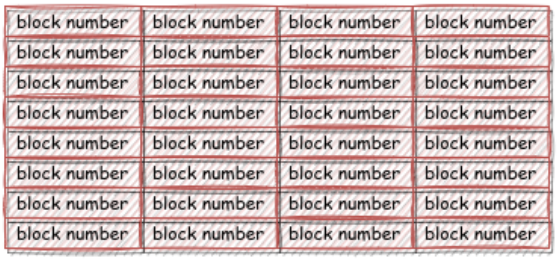
Boot, Free



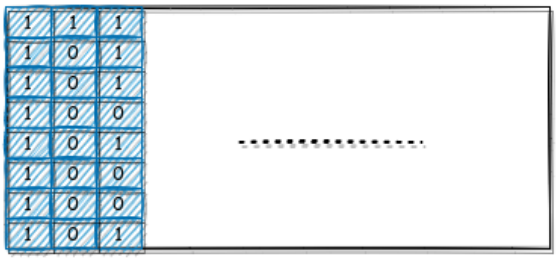
Super



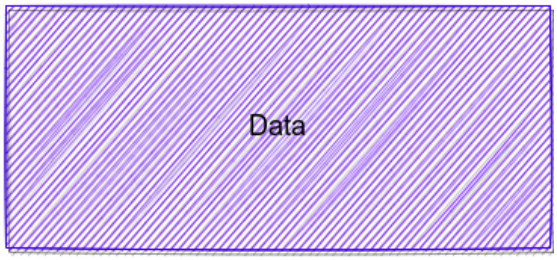
File



Index



BMap

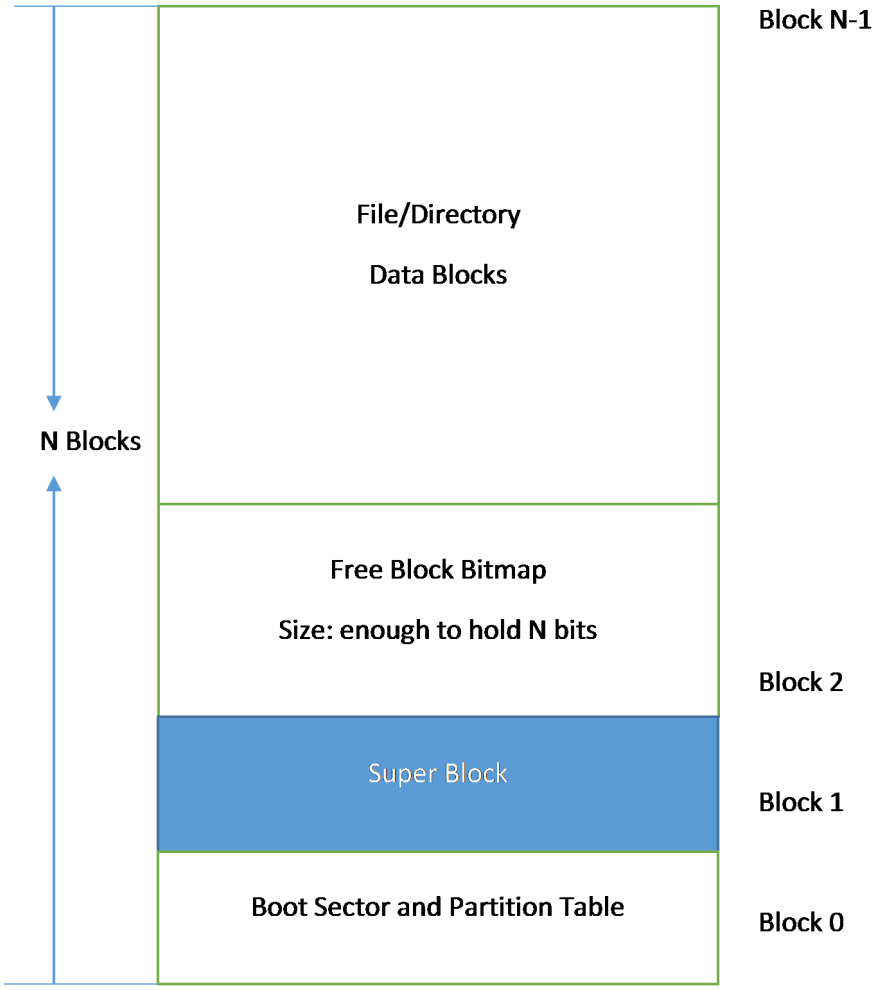


Data

*Part*2.2.1 磁盘文件系统布局

磁盘结构和初始化

磁盘空间的基本布局如下图所示



图中出现的 Block 是磁盘块。不同于扇区 Sector，**磁盘块是一个虚拟概念，是操作系统与磁盘交互的最小单位**；操作系统将相邻的扇区组合在一起，形成磁盘块进行整体操作，减小了因扇区过多带来的寻址困难；磁盘块的大小由操作系统决定，一般由2的n次方个扇区构成。而**扇区是真实存在的，是磁盘读写的基本单位，与操作系统无关**。（512 = 0x200）

在我们的实验中，采用8个扇区组成一个磁盘块的方式(8*512 = 4096B = 4KB)。由于这个大小和之前的内存一页的大小是相同的，所以真实的Linux操作系统中大多数的磁盘块也是由8个扇区组成的4K大小。

从图中可以看到，MOS 操作系统把磁盘最开始的一个磁盘块 (4096 字节) 当作启动扇区和分区表使用。接下来的一个磁盘块作为超级块 (Super Block)，用来描述文件系统的基本信息，如 Magic Number、磁盘大小以及根目录的位置。

> Note：在真实的文件系统中，一般会维护多个超级块，通过复制分散到不同的磁盘分区中，以防止超级块的损坏造成整个磁盘无法使用。

MOS 操作系统中超级块的结构如下

```
struct Super
{
    u_int s_magic;      // Magic number: FS_MAGIC
    u_int s_nblocks;    // Total number of blocks on disk
    struct File s_root; // Root directory node
};
struct Super super; // super block.
```

其中每个域的意义如下：

- **s_magic**：魔数，用于识别该文件系统，为一个常量。

> 这里会放入一个宏 #define FS_MAGIC 0x68286097
>
> > 这里是一个彩蛋，这个宏的值是我们北航OS课程的课程代码。

- **s_nblocks**：记录本文件系统有多少个磁盘块，本文件系统为 1024。

> 宏也定义了这一数值：#define NBLOCK 1024

- **s_root**：根目录，其f_type为FTYPE_DIR，f_name为"/"。

通常采用两种数据结构来管理可用的资源：链表和位图。

在 lab2 和 lab3 实验中，我们使用了链表来管理空闲内存资源和进程控制块。在文件系统中，我们将使用位图 (Bitmap) 法来管理空闲的磁盘资源，用一个二进制位 bit 标识磁盘中的每个磁盘块的使用情况（注意：**实验中，1 表示空闲**）。

这里我们参考 tools/fsformat 表述文件系统标记空闲块的机制。tools/fsformat 是用于创建符合我们定义的文件系统结构的工具，用于将多个文件按照内核所定义的文件系统写入到磁盘镜像中。在写入文件之前，fs/fsformat.c 的init_disk函数，将所有的块都标为空闲块：

```
    nbitblock = (NBLOCK + BIT2BLK - 1) / BIT2BLK;
    for(i = 0; i < nbitblock; ++i) {
        memset(disk[2+i].data, 0xff, BY2BLK);
    }
    if(nblock != nbitblock * BIT2BLK) {
        diff = nblock % BIT2BLK / 8;
        memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
    }
```

nbitblock 表示记录整个磁盘上所有块的使用信息，需要多少个磁盘块来存储位图。 紧接着，我们使用 memset 将位图中的每一个字节 (Byte) 都设成 0xff ，即将所有位图块的每一位都设为 1 ，表示这一块磁盘处于空闲状态。如果位图还有剩余，不能将最后一块位图块中靠后的一部分内容标记为空闲，因为这些位所对应的磁盘块并不存在，不可被使用。因此，将所有的位图块的每一位都置为 1 之后，还需要根据实际情况，将位图不存在的部分设为0。

init_disk() 初始化磁盘

这里吐槽一句：这里 nbitblock 的计算用到了$\lceil \frac{a}{b} \rceil = \lfloor \frac{a+b-1}{b} \rfloor$。

```
// Initial the disk. Do some work with bitmap and super block.
void init_disk() {
    int i, r, diff;

    // Step 1: Mark boot sector block.
    /* 将第0个磁盘块作为启动磁盘块 */
    disk[0].type = BLOCK_BOOT;  // BLOCK_BOOT  = 1

    // Step 2: Initialize boundary.
    /* 计算出需要多少个磁盘块来存储位图 */
    nbitblock = (NBLOCK + BIT2BLK - 1) / BIT2BLK;
    /* 这里其实是NBLOCK除以BIT2BLK的上取整。
    总共的磁盘块数量 除以 一个块的位(bit)数的上取整，就是
    表示整个磁盘的磁盘块的 位图 需要多少个磁盘块 来表示 */
    /*  (1024 + 4096*8 - 1) / (4096*8) = 1*/
    nextbno = 2 + nbitblock;
    // 算出存放位图的起始磁盘块序号+1的值，因为第0个是启动磁盘块，第1个是超级磁盘块

    // Step 2: Initialize bitmap blocks.
    /* 初始化存储位图的磁盘块（简称位图磁盘块）*/
    for(i = 0; i < nbitblock; ++i) {
```

```
        disk[2+i].type = BLOCK_BMAP;  // BLOCK_BMAP  = 2
    }
    /* 将位图磁盘块 按字节(8bits, uint_8)置为0xff，即每一位都置1，
    表示现在所有的磁盘块都是空闲的 */
    for(i = 0; i < nbitblock; ++i) {
        memset(disk[2+i].data, 0xff, BY2BLK);
    }
    /* 如果说磁盘块的个数（即需要占用位图来描述的位数）不等于当前占用的位图磁盘块的总位数-->说
明有剩余 */
    if(NBLOCK != nbitblock * BIT2BLK) {
        diff = NBLOCK % BIT2BLK / 8;// diff是位位图的在最后一块位图磁盘块中实际占据的大
小（字节）
        /* 计算位图中剩余的部分，这部分位图剩余对应着磁盘上不存在的部分，因此需要置为0，表示不
可以使用。
        1024 / 8 = 128 这部分位图是对应磁盘上存在的部分，那么剩下的都要置0。
        起点便是 diff，置零长度为 BY2BLK - diff */
        memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
        # 这里转为字节主要是因为memset只能以一字节为单位进行赋值
    }

    // Step 3: Initialize super block.
    disk[1].type = BLOCK_SUPER; // BLOCK_SUPER = 3
    /* 这里的super 就是 Struct Super类型的结构体，即超级磁盘块 */
    super.s_magic = FS_MAGIC;
    super.s_nblocks = NBLOCK; // 1024
    super.s_root.f_type = FTYPE_DIR;
    strcpy(super.s_root.f_name, "/");// 设置超级块的文件名是"/"
}
```

相应地，在 MOS 操作系统中，文件系统也需要根据位图来判断和标记磁盘的使用情况。fs/fs.c 中的 block_is_free 函数就用来通过位图中的特定位来判断指定的磁盘块是否被占用。

当然，这里的位图就是由全局变量bitmap指向的位图磁盘块。通过bitmap[]数组寻址的方式和bitmap本身u_int *的类型，就可以在位图控制块结构体中数据部分，以4字节为单位进行查询。

block_is_free()

利用位图检查磁盘块序号为 blockno 的磁盘块是否是空闲状态。

如果是处于空闲状态，则返回1，否则返回0。

```
int block_is_free(u_int blockno)
{
    if (super == 0 || blockno >= super->s_nblocks) {
        return 0;
    }
    // 经典的整除32和模运算32，位图管理法
    // 磁盘块号 blockno / 32 的商说明是第几个32位整数上（从第0个开始），
    // blockno % 32 的余数则是在这个32位整数上的第几位
    // 下面这个与运算就是检查位图中这一位代表的 磁盘块（也就是磁盘块号 blockno 所代表的这一块
磁盘块）是否空闲，如果是1，则说明是空闲的
    if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
        return 1;
    }
    return 0;
}
```

文件系统需要负责维护磁盘块的申请和释放，在回收一个磁盘块时，需要更改位图中的标志位。如果要将一个磁盘块设置为free，只需要将位图中对应的位的值设置为1 即可。

fs/fs.c 中的 free_block 函数就是用于实现这一功能。同时参数 blockno 的值不能为0——因为第0个磁盘块是用作磁盘的启动磁盘块。

free_block()

借助位图将磁盘块号 blockno 所代表的磁盘块标记为空闲。

```
void free_block(u_int blockno)
{
    // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be
zero).
    if (blockno == 0 || blockno >= super->s_nblocks) {
        user_panic("blockno is zero");
    }
    // Step 2: Update the flag bit in bitmap.
    // you can use bit operation to update flags, such as  a |= (1 << n) .
    bitmap[blockno / 32] |= (1 << (blockno % 32));
}
```

$Part2.2.2$ 文件系统详细结构

同样的，先介绍一些宏。**这些分析很重要！**

```
// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN  128
// Maximum size of a complete pathname, including null
#define MAXPATHLEN  1024
// Number of (direct) block pointers in a File descriptor
#define NDIRECT     10
/* Number of block pointers in a Block  */
// 由于间接磁盘块指针f_indirect的存在，一个文件控制块最多可以有一个磁盘块大小的指向磁盘块的指针，一个指针的大小是4个字节（就是磁盘块号，u_int类型）。那么一个文件控制块最多有 BY2BLK/4 = 1024个指向磁盘块的指针。
// 当这个文件控制块的类型f_type是目录文件FTYPE_DIR时，这意味着，一个目录里面最多下辖1024个子文件/子目录
// 也就是这里的 NINDIRECT = (BY2BLK/4)
// 当这个文件控制块的类型f_type是普通文件FTYPE_REG时，也就说这1024个它指向的磁盘块存放的全是它的文件内容。因此就不难得出：单个文件最大为 1024*BY2BLK = 4M；也可以理解为，单个文件最多占用1024个磁盘块。
// 也就是这里的 MAXFILESIZE = NINDIRECT * BY2BLK
#define NINDIRECT   (BY2BLK/4)
#define MAXFILESIZE (NINDIRECT*BY2BLK)

#define BY2FILE     256 // bytes to file 文件控制块 struct File 的字节数
#define FILE2BLK    (BY2BLK/sizeof(struct File))
// files to Block 一个磁盘块最多放置的文件控制块个数，其实就是 4K / 256 = 16 ；sizeof(struct File)也就是 BY2FILE （所以为什么不直接 BY2BLK / BY2FILE ）
```

File结构体

操作系统要想管理一类资源，就得有相应的数据结构。对于描述和管理文件来说，一般使用文件控制块（File 结构体）。其定义如下：

```
// file control blocks, defined in include/fs.h
struct File
{
    /* 除了最后一个成员 其他成员的字节数 MAXNAMELEN + 4 + 4 + NDIRECT*4 + 4 */
```

```
    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; // valid only in memory
    u_char f_pad[256 - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
    /* 由此可以得出，一个文件控制块总共有 256 个字节 */
};
```

结合文件控制块的示意图，我们对各个域进行解读：

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。

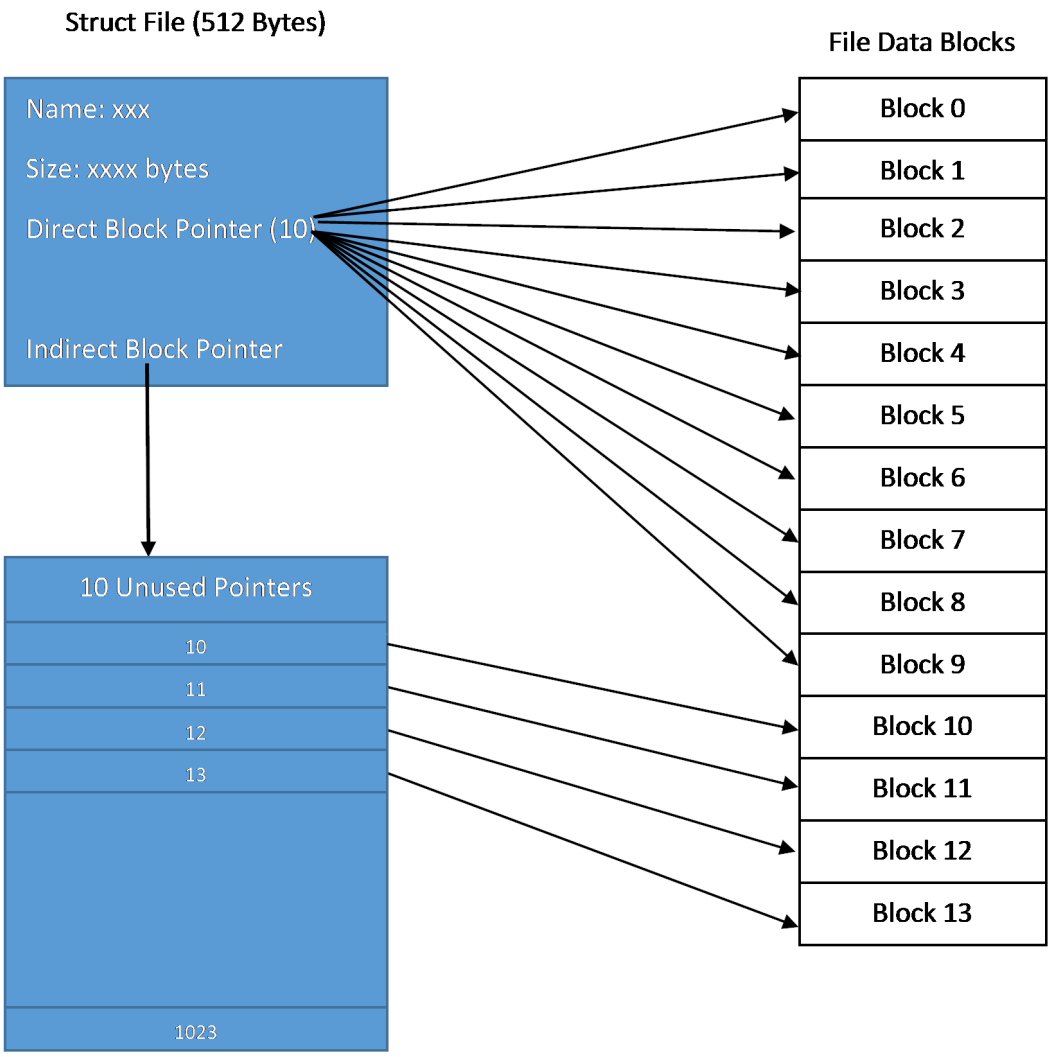
f_direct[NDIRECT] 为文件的直接指针，每个文件控制块设有 10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4KB，也就是说，这十个直接指针能够表示最大 40KB 的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。f_indirect指向一个间接磁盘块，用来存储指向文件内容的磁盘块的指针。为了简化计算，我们不使用间接磁盘块的前十个指针。

f_dir指向文件所属的文件目录。**即指向一个类型为文件夹 (FTYPE_DIR) 的文件控制块**。

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。

再次强调，这里的“指针”并不是指针类型的变量，而是相应的磁盘块在disk数组中的下标。因此，文件控制块中，用于存储这些“指针”的相应变量f_direct[NDIRECT] 和 f_indirect 才会是 u_int 类型。

> Note：我们的文件系统中的文件控制块只使用了一级间接指针域，也只有一 个。而在真实的文件系统中，为了支持更大的文件，通常会使用多个间接磁盘块， 或使用多级间接磁盘块。MOS 操作系统内核在这一点上做了极大的简化。



对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录所包含的文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。

> 为了更加细致地了解文件系统的内部结构，我们通过 fsformat（由 fs/fsformat.c 编译而成）程序来创建一个磁盘镜像文件 gxemul/fs.img。通过观察头文件和 fs/Makefile，我们可以看出，fs/fsformat.c 的编译过程与其他文件有所不同，其使用的是 Linux 下的gcc 编译器，而非 mips_4KC-gcc 交叉编译器。编译生成的 fsformat 独立于 MOS 操作系统，专门用于创建磁盘镜像文件。生成的镜像文件 fs.img 可以模拟与真实的磁盘文件设备的交互。请阅读 fs/fsformat.c 和 fs/Makefile，掌握如何将文件和文件夹按照文件系统的格式写入磁盘，了解文件系统结构的具体细节，学会添加自定义文件进入磁盘镜像。（fsformat.c 中的主函数十分灵活，可以通过修改命令行参数来生成不同的镜像文件）

向磁盘镜像中烧录文件

虽然本质上磁盘镜像只是一个“数据生成器”，但是通过研究文件的生成，可以很好的理解文件系统的工作机制。

next_block()

这个函数用于获得一个空闲的块，并且将这个块的属性进行设置。

通过全局变量 `nextbno` 作为索引，将磁盘disk数组中下标为 `nextbno` 的磁盘块的类型设置为参数type，并且**先返回 `nextbno` ，再将 `nextbno` 加1。**

```
uint32_t nextbno;   // next availiable block.
// Get next block id, and set `type` to the block's type.
int next_block(int type) {
    disk[nextbno].type = type;
    return nextbno++;
}
```

save_block_link()

把第 bno 个磁盘块 disk[bno] 的磁盘块号码 bno 存放在文件控制块 f 的第 nblk 个指针里。也可以概括地说：这个函数用于在建立一个文件控制块体与特定的磁盘块之间的联系。因为文件控制块里会有很多个指针指向磁盘块。

显然，$\mathrm{nblk} \in [0, 1023]$。

> nblk 是在文件中的第几个指针，反应的是**该磁盘块在文件中的编号**，而bno则是该磁盘块在整个磁盘中的序号。
>
> 比如某个文件的第一个磁盘块在磁盘中的编号是 `301` ，但是他是该文件的第一个磁盘块，所以有 `nblk = 0, bno = 301`

`f->f_indirect == 0` 的判断要留意：间接“指针”存放的值是0，说明此前没有给file文件控制块分配磁盘块用于存放它下辖的磁盘块的号码。因为第0个磁盘块 disk[0] 是固定为磁盘启动，不可以被使用。

```
// Save block link.
void save_block_link(struct File *f, int nblk, int bno)
{
    // 单个文件最多占用1024个磁盘块，如果比这个多，显然文件太大了
    assert(nblk < NINDIRECT); // if not, file is too large !
    // 如果要映射保存到的 在当前文件下辖的磁盘块号 小于10，那么将此磁盘块的指针地址（也就是在整个disk[]数组中的磁盘块号）直接存入 文件file的直接指针数组f_direct[NDIRECT]
    if(nblk < NDIRECT) {
        f->f_direct[nblk] = bno;
    }
    // 要存到file文件控制块中的磁盘块号>=10，就要存到间接“指针”指向的存放磁盘块的号码的磁盘
    else {
        if(f->f_indirect == 0) {// 间接“指针”存放的值是0，说明此前没有给file文件控制块分配磁盘块用于存放它下辖的磁盘块的号码。因为第0个磁盘块 disk[0] 是固定为磁盘启动，不可以被使用。
            // create new indirect block.
            f->f_indirect = next_block(BLOCK_INDEX);
            // 返回 nextbno ，将 nextbno 指向的磁盘块类型设置为 BLOCK_INDEX ，最后将 nextbno 加1
            /* 由此可以看出，类型为 BLOCK_INDEX 的磁盘块存放的就是一个个磁盘块序号，被存放了磁盘块序号的这些磁盘块的类型则是 BLOCK_FILE */
        }
        // 将存放间接“指针”的磁盘块作为“指针”数组，令第nblk个“指针”的值设置为bno
        /// 其实这个磁盘块就是存放磁盘块号码的
        ((uint32_t *)(disk[f->f_indirect].data))[nblk] = bno;
    }
}
```

make_link_block()

这个函数会为一个目录分配出一个新的磁盘块来存储目录中包含的文件控制块结构体

具体来说，这个函数将当前 disk[nextbno] 对应的磁盘块的磁盘块号 nextbno 存入 文件控制块 dirf 的第 nblk 个指针中。

返回 nextbno 的值，并让 nextbno加1，文件控制块 dirf 的f_size加上BY2BLK。

其实也就是分配一个磁盘块作为 **文件** 归属于文件控制块 dirf。

```
// Make new block contians link to files in a directory.
int make_link_block(struct File *dirf, int nblk) {
    // 把nextbon所指向的磁盘块的类型置为`文件类型` BLOCK_FILE ，并将nextbno赋给bno后，
nextbno加1
    int bno = next_block(BLOCK_FILE);// BLOCK_FILE  = 5
    save_block_link(dirf, nblk, bno);
    dirf->f_size += BY2BLK;
    return bno;
}
```

create_file()

这个函数会在指定的的目录 dirf 下新创建一个文件。具体的原理就是查找目录文件的磁盘块还有没有空余位置，有的话就放入一个文件控制块，要是没有的话，就利用 `make_link_block` 创建一个。

具体来说：寻找文件控制块 dirf 所存储的磁盘块号（即“磁盘块指针”）对应的磁盘块中，有没有空闲的文件控制块。如果有则返回，如果没有，则让文件控制块 dirf 多管理一个磁盘块（nextbno代表的磁盘块），并返回新的磁盘块的第0个文件控制块的地址。

```
// Overview:
//      Create new block pointer for a file under sepcified directory.
//      Notice that when we delete a file, we do not re-arrenge all
//      other file pointers, so we should be careful of existing empty
//      file pointers
//
// Post-Condition:
//      We ASSUME that this function will never fail
//
// Return:
//      Return a unused struct File pointer
// Hint:
//      use make_link_block function
struct File *create_file(struct File *dirf) {
    struct File *dirblk;
    int i, j, bno, found;
    /* 计算出文件内容占用的磁盘块数目。 */
    int nblk = dirf->f_size / BY2BLK;

    // Step1: According to different range of nblk, make classified discussion
to
    //        calculate the correct block number.
    for (i = 0; i < nblk; i++) {// 文件占用的第 i 个磁盘块
        if (i < NDIRECT) {
            bno = dirf->f_direct[i];// 取出直接指针存放的磁盘号码
        } else {
            bno = ((uint32_t *)(disk[dirf->f_indirect].data))[i];
            // 取出间接指针存放的磁盘号码
        }
        // 现在bno对应的磁盘块disk[bno]肯定不是存放指针的磁盘块 BLOCK_INDEX，因为本实验是
一级间接指针域。所以都是 BLOCK_FILE 类型的，直接 将磁盘内容用文件控制块的结构体形式表示，那么
最多有4K/256 = 16 = FILE2BLK个
```

```
            // 在遍历每个磁盘块的时候，步长改为文件控制块大小。
            dirblk = (struct File *)(disk[bno].data);
            // 遍历每一个磁盘块存放的16个文件磁盘块。
            for (j = 0; j < FILE2BLK; j++) {
                // 在第i个磁盘块中，发现它的第j个文件控制块是空的，就可以返回它
                if (dirblk[j].f_name[0] == '\0') {
                    return &dirblk[j];
                }
            }
        }
        // 在当前文件已经占用的nblk个磁盘块(0 ~ nblk-1)中找不到空闲的文件控制块
        // 那么就需要新下辖一个磁盘块，即文件控制块 f 的第 nblk 个磁盘指针中存入 nextbno 的值
        // Step2: Find an unused pointer
        bno = make_link_block(dirf, nblk);
        // 这个函数将当前 disk[nextbno] 对应的磁盘块的磁盘块号 nextbno 存入 文件控制块 dirf
的第 nblk 个指针中。
        return (struct File *)disk[bno].data;// 返回新下辖的磁盘的第0个文件控制块
    }
```

write_file()

这个函数实现的功能就是把一个已有的文件写入到我们的磁盘镜像中。要写入到的目录是 dirf，要写入到的路径是path

> 这个函数就偏向于构造镜像了，跟磁盘文件系统的关系就比较小了，难点也是理解已有的库函数。

```
// Write file to disk under specified dir.
void write_file(struct File *dirf, const char *path) {
    int iblk = 0, r = 0, n = sizeof(disk[0].data);
    uint8_t buffer[n+1], *dist;
    // 首先我们在目录 dirf 下创造一个空文件，其实就是获得了一个文件控制块
    struct File *target = create_file(dirf);

    /* in case `create_file` is't filled */
    if (target == NULL) return;

    int fd = open(path, O_RDONLY);

    // Get file name with no path prefix.
    // 利用各种系统调用把各种文件的属性都登记在这个文件控制块里
    const char *fname = strrchr(path, '/');
    if(fname)
        fname++;
    else
        fname = path;
    strcpy(target->f_name, fname);

    target->f_size = lseek(fd, 0, SEEK_END);
    target->f_type = FTYPE_REG;

    // Start reading file.
    // 然后开始一个块一个块的烧录
    lseek(fd, 0, SEEK_SET);
    while((r = read(fd, disk[nextbno].data, n)) > 0) {
        save_block_link(target, iblk++, next_block(BLOCK_DATA));
    }
    close(fd); // Close file descriptor.
}
```

Part2.2.3 块缓存

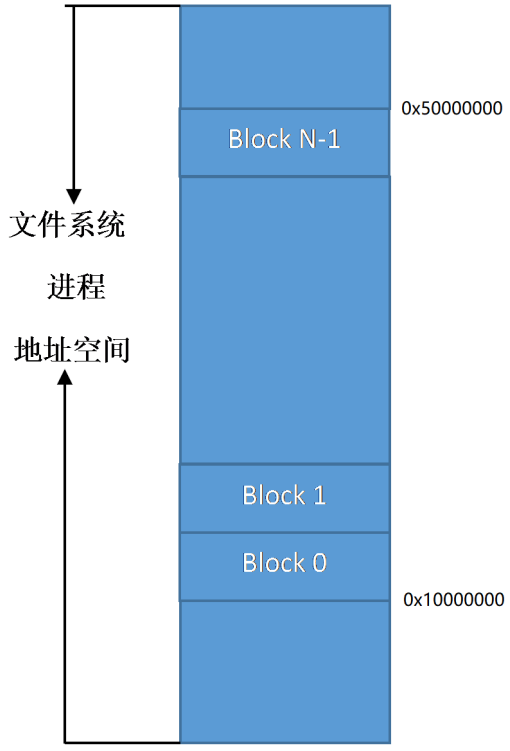
块缓存指的是借助虚拟内存来实现磁盘块缓存的设计。我们的操作系统中，文件系统服务是一个用户进程（将在下文介绍），一个进程可以拥有4G 的虚拟内存空间，将DISKMAP 到 DISKMAP+DISKMAX 这一段虚存地址空间(0x10000000-0x4fffffff) 作为缓冲区，当磁盘读入内存时，用来映射相关的页。

注意缓冲区的大小是1G而不是4G。

DISKMAP 和 DISKMAX 的值定义在 fs/fs.h 中：

```
/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP     0x10000000

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

为了建立起磁盘地址空间和进程虚存地址空间之间的缓存映射，我们采用的设计如下图所示。



> 当完成了 `ide_read, ide_write` 之后，似乎就已经完成了与磁盘文件系统的对接工作，我们已经可以着手构建上层操作系统文件系统的工作了。但是实际上并没有，我们又引入了“块缓存”机制。这个机制我理解了很久，是因为它干的事情就是把磁盘中的内容先拷贝到一个固定虚拟地址空间，当我们访问磁盘的时候，并不直接访问磁盘，而是访问这片地址空间，这种莫名其秒的“中转行为”让我理解了很久。
>
> 但是其实是因为我没有考虑到读写磁盘的耗时问题，如果只使用 `ide_read, ide_write` 。那么相当于每次都需要读写磁盘，而读写磁盘是一个耗时的活动，所以就会造成大量的时间浪费，设置一个 cache 一样的结构，就可以提高读写效率。这就是块缓存存在的意义。
>
> 另外可以发现，如果将内存中的一部分（ `DISKMAP ~ DISKMAP + DISKMAX` ）当作 cache 来看，有两点有趣的事情：
>
> - 这片区域是比一个磁盘大的，cache 到磁盘的映射规则是直接映射，而且一定不会发生冲突，也不支持发生冲突（现有 MOS）
> - 这片区域是在用户地址空间的，但是并不会出现一致性问题，因为只有一个用户进程有块缓存机制，那就是文件系统服务进程。
>
> 这个部分的实现在 `fs.c` 的前面位置。

块缓存实现——block相关的函数解读

与 cache 类似，我们需要一系列的函数来是实现块缓存的功能，比如检测某个块在不在缓存区内，写回的时候需不需要拷贝（以脏没脏没基准），总之这些都需要实现，实现起来还是很简单的

diskaddr()

fs/fs.c 中的 diskaddr 函数用来计算指定磁盘块对应的虚存地址。

根据一个块的序号bno (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。

核心公式：$\mathrm{DISKMAP} + \mathrm{blockno} * \mathrm{BY2BLK};$

```
// Overview:
//  Return the virtual address of this disk block. If the `blockno` is greater
//  than disk's nblocks, panic.
u_int diskaddr(u_int blockno)
{
    if (super && blockno >= super->s_nblocks) {// 这里的super->s_nblocks就是NBLOCK
= 1024
        user_panic("reading non-existent block %08x\n", blockno);
    }
    return DISKMAP + blockno * BY2BLK;
}
```

va_is_mapped() 检查虚拟地址va是否已经映射了一个磁盘块

这个函数用于检查（内核空间和进程空间的）虚拟地址va是否已经映射了一个磁盘块，其实就是检查：va所对应的一级页表项和二级页表项是否有效(check PTE_V bit)，也就是检查：是否在va对应的二级页表项中已经存入了 为了保存磁盘块数据而分配的物理内存 的物理页号+权限位。

```
// Overview:
//  Check if this virtual address is mapped to a block. (check PTE_V bit)
u_int va_is_mapped(u_int va)
{   // 用PDX(va)得到是在第几个一级页表项，用VPN(va)得到虚拟页号，就是第几个二级页表项
    return (((*vpd)[PDX(va)] & (PTE_V)) && ((*vpt)[VPN(va)] & (PTE_V)));
}
```

复习lab4中关于vpt和vpd的总结：

> 可以发现，全局变量vpt的值是 `UVPT`（`0x7fc0 0000`），也就是当前进程空间中二级页表区的首地址。vpt的变量类型为Pte*，所以可以通过$(*vpt)[index], index \in [0, 1024 * 1024 - 1]$范围内的下标进行数组寻址（1024个二级页表打破彼此的隔阂），找到任意一个进程空间中的二级页表区的二级页表项。进而通过这个页表项，按页为单位找到4G空间。
>
> 而全局变量vpd的值是 `(UVPT+(UVPT>>12)*4)`，也就是当前进程空间中用于表示二级页表区的一级页表的基地址。vpd的变量类型为Pde*，所以可通过$(*vpd)[index], index \in [0, 1023]$范围内的下标进行数组寻址，找到任意一个一级页表项，每一个一级页表项又代表一个二级页表区中的二级页表。

block_is_mapped() 检查磁盘块是否已经映射/加载到了物理内存和进程虚拟空间（即这个磁盘块已经实现了块缓存）

检查序号为 `blockno` 的磁盘块是否已经映射到了物理内存和进程虚拟空间（通过使用 `diskaddr(blockno)` 得到对应的虚拟地址 va，再用 `va_is_mapped(va)` 检查 va 是否已经建立映射）

是，则返回 va；否，则返回 0。

```
// Overview:
//  Check if this disk block is mapped to a vitrual memory address. (check
corresponding `va`)
u_int block_is_mapped(u_int blockno)
{
    u_int va = diskaddr(blockno);
    if (va_is_mapped(va)) {
        return va;
    }
    return 0;
}
```

当我们把一个磁盘块(block) 中的内容载入到内存中时，我们需要为之分配对应的物理内存，当结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。fs/fs.c 中的map_block 函数和unmap_block 函数实现了这一功能。

注意：map_block() 和unmap_block() 两个函数仅仅是建立和取消指定磁盘块 `blockno` 对应的虚拟内存地址 `diskaddr(blockno)` 和实际物理内存的映射关系（以及在必要时申请和释放物理内存），并不是加载磁盘块内容到虚拟内存和物理内存中。这个功能要靠read/write_block()实现。

map_block()

为一个指定的磁盘块 在 内存中该磁盘块对应的地址 `diskaddr(blockno)` 处 建立映射关系。即：申请对应的物理空间，填写对应页表项。

检查序号为 blockno 的磁盘块 disk[blockno]是否已经和物理内存、进程虚拟空间建立了映射，如果没有，则为当前进程的**进程空间下的该磁盘块对应的虚拟地址，申请一页实际的物理内存用于保存磁盘块上的数据**，并将这页物理内存的物理页号+权限（PTE_V | PTE_R）存放在当前进程页目录映射的二级页表体系中对应的二级页表项中（当然对应的二级页表也是需要申请物理内存的）。

```
int map_block(u_int blockno)
{
    // Step 1: Decide whether this block has already mapped to a page of
physical memory.
    if (block_is_mapped(blockno)) {
        return 0;
    }
    // Step 2: Alloc a page of memory for this block via syscall.
    return syscall_mem_alloc(0, diskaddr(blockno), PTE_V | PTE_R);
}
```

unmap_block()

解除序号为 blockno 的磁盘块 disk[blockno] 和物理内存的映射关系，回收内存。

这里主要是将进程空间中序号为 blockno 的磁盘块对应的虚拟地址va对应的进程页目录映射的二级页表体系的二级页表项中的物理页号和权限位清零，顺便减少物理页面引用次数，看情况释放物理页面。做完这些后更新TLB。

```
void unmap_block(u_int blockno)
{
    int r;
    u_int addr;
    // Step 1: check if this block is mapped.
```

```
    // block_is_mapped(blockno)返回值为0，说明序号为 blockno 的磁盘块还没有建立映射
    addr = block_is_mapped(blockno);
    if (addr == 0) {
        return;
    }
    // Step 2: use block_is_free, block_is_dirty to check block,
    // if this block is used(not free) and dirty, it needs to be synced to disk:
write_block
    // can't be unmap directly.
    if (!block_is_free(blockno) && block_is_dirty(blockno)) {
        write_block(blockno);
    }
    // Step 3: use 'syscall_mem_unmap' to unmap corresponding virtual memory.
    r = syscall_mem_unmap(0, addr);
    if (r < 0) {
        user_panic("unmap_block failed");
    }
    // Step 4: validate result of this unmap operation.
    user_assert(!block_is_mapped(blockno));// 检查序号为 blockno 的磁盘块是否已经解除
了映射
}
```

读写磁盘块

read_block 函数和 write_block 函数用于读写磁盘块。

```
/* IDE disk number to look on for our file system */
#define DISKNO      1

#define BY2SECT     512 /* Bytes per disk sector */
#define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */
// 一个磁盘块拥有的扇区数目
```

read_block()

将指定编号的磁盘块读入到内存中。

将传入的参数 blockno 代表的磁盘块读入到内存中。首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 ide_read 函数来读取磁盘上的数据到对应的虚存地址处。

- 如果说参数blk != 0，就设置参数 *blk 为 blockno 代表的磁盘块在**虚拟内存中的地址**。
- 如果说参数isnew != 0
    - 并且 blockno 代表的磁盘块早就已经读入到了物理内存和进程虚拟空间中，就设置isnew = 0；
    - 并且 blockno 代表的磁盘块还没有读入到物理内存和进程虚拟空间中，就设置isnew = 1。然后为当前进程的进程空间下的该磁盘块对应的虚拟地址 `va = diskaddr(blockno)` 申请一页实际的物理内存用于保存磁盘块上的数据，再将该磁盘块上的数据加载到物理内存和进程虚拟空间；

根据该函数的oveview，这个函数一个重要的作用是 **确保某个特定的磁盘块已经被加载到了物理内存和用户进程空间。具体体现在参数 *isnew 是0 还是1，如果是0，则说明磁盘块数据内容已经加载到了内存。**

```
// Overview:
//  Make sure a particular disk block is loaded into memory.
//
// Post-Condition:
```

```
//  Return 0 on success, or a negative error code on error.
//
//  If blk!=0, set *blk to the address of the block in memory.
//
//  If isnew!=0, set *isnew to 0 if the block was already in memory, or
//  to 1 if the block was loaded off disk to satisfy this request. (Isnew
//  lets callers like file_get_block clear any memory-only fields
//  from the disk blocks when they come in off disk.)
//
// Hint:
//  use diskaddr, block_is_mapped, syscall_mem_alloc, and ide_read.
int read_block(u_int blockno, void **blk, u_int *isnew)
{
    u_int va;
    // Step 1: validate blockno. Make file the block to read is within the disk.
    /* 检验 blockno 的有效性，如果super块指针不为空，且blockno超过了总共磁盘块数目，那么就说明读到了不存在的磁盘块，报错 */
    if (super && blockno >= super->s_nblocks) {
        user_panic("reading non-existent block %08x\n", blockno);
    }
    // Step 2: validate this block is used, not free.
    /* 检验 blockno 代表的磁盘块是否是空闲的，如果是，则报错（因为我们要读出一个已经使用了的磁盘块）
        ! 这里值得注意的是：
        每当我们使用 block_is_free(blockno) 函数去检查 blockno 代表的磁盘块是否是空闲磁盘块的时候，我们必须确保全局变量 bitmap!=NULL，即我们已经把 位图磁盘块 从磁盘中读入了内存。
        bitmap就是指向 位图磁盘块的指针
    */
    if (bitmap && block_is_free(blockno)) {
        user_panic("reading free block %08x\n", blockno);
    }
    // Step 3: transform block number to corresponding virtual address.
    // 根据 blockno 计算出 该磁盘块应该被加载到的 512B内存虚拟空间的 基地址
    va = diskaddr(blockno);
    // Step 4: read disk and set *isnew.
    // 如果 blockno 对应的磁盘块已经映射到了内存（即加载到了内存中）
    // 那么只在 isnew 不为0时，设置 *isnew 为0（说明不是新的磁盘块）
    // 否则就在 isnew 不为0时，设置 *isnew 为1（说明是新的磁盘块）
    // 并且为新的磁盘块申请一页物理内存（PTE_V | PTE_R），并将磁盘块读到 内存中 上一步计算出的va处
    // 我们只有一个磁盘，所以 使用 ide_read 函数时，磁盘号应该是 0
    // Hint: if this block is already mapped, just set *isnew, else alloc memory and
    // read data from IDE disk (use `syscall_mem_alloc` and `ide_read`).
    // We have only one IDE disk, so the diskno of ide_read should be 0.
    if (block_is_mapped(blockno))
    { //the block is in memory
        if (isnew)
            *isnew = 0;
    }
    else
    { //the block is not in memory
        if (isnew)
            *isnew = 1;
        syscall_mem_alloc(0, va, PTE_V | PTE_R);
        /* 为当前进程的进程空间下的该磁盘块对应的虚拟地址 va = diskaddr(blockno)
           申请一页实际的物理内存用于保存磁盘块上的数据，
           并将这页物理内存的物理页号+权限（PTE_V | PTE_R）存放在
           当前进程页目录映射的二级页表体系中对应的二级页表项中 */
        ide_read(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
        /* 从0号磁盘的第 blockno * SECT2BLK 个扇区开始，将 SECT2BLK 个扇区（也就是一整个磁盘块）
           的磁盘内容读入到 虚拟内存空间 的 va 处*/
    }
    // Step 5: if blk != NULL, set `va` to *blk.
    if (blk)
```

```
        *blk = (void *)va;
    return 0;
}
```

write_block()

将序号为 blockno 的磁盘块 disk[blockno] 对应的**虚存空间内中整个磁盘块的数据** 写入到 **磁盘块** 中。

```
void write_block(u_int blockno)
{
    u_int va;
    // Step 1: detect is this block is mapped, if not, can't write it's data to
disk.
    // 检查序号为 blockno 的磁盘块是否已经加载到了物理内存
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }
    // Step2: write data to IDE disk. (using ide_write, and the diskno is 0)
    va = diskaddr(blockno);
    // 选择0号磁盘，将 va 代表的地址开始 SECT2BLK 个扇区（也就是一整个磁盘块）的数据，写入到
    // 0号磁盘的 从第 blocknoo * SECT2BLK个扇区开始 的 SECT2BLK(8) 个扇区中
    ide_write(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
    // 这里由于同一个进程的同一个地址，所以这里的syscall_mem_map调用的page_insert其实只是
    // 更新tlb 和 更新pp对应物理页面基地址的权限（因为在lab4里面会给父进程的页表项新增
PTE_COW用于保护，lab5里面会新增PTE_LIBRARY）
    syscall_mem_map(0, va, 0, va, (PTE_V | PTE_R | PTE_LIBRARY));
}
```

读写磁盘块辅助函数

alloc_block_num()

分配磁盘块号：在位图中寻找一个空闲的磁盘块，如果成功，返回找到的这个磁盘块的序号 blockno。

优先分配序号小的磁盘块。分配的磁盘块号在[3, super->s_nblocks-1]之间（即 $[3, 1023]$ ）

```
// Overview:
//  Search in the bitmap for a free block and allocate it.
//
// Post-Condition:
//  Return block number allocated on success,
//         -E_NO_DISK if we are out of blocks.
int alloc_block_num(void)
{
    int blockno;
    // walk through this bitmap, find a free one and mark it as used, then sync
    // this block to IDE disk (using `write_block`) from memory.
    for (blockno = 3; blockno < super->s_nblocks; blockno++) {
        if (bitmap[blockno / 32] & (1 << (blockno % 32))) { // the block is free
            bitmap[blockno / 32] &= ~(1 << (blockno % 32));
            write_block(blockno / BIT2BLK + 2); // write to disk.
            return blockno;
        }
    }
    // no free blocks.
    return -E_NO_DISK;
}
```

alloc_block()

分配磁盘块：借助 `alloc_block_num()` 函数在位图中寻找一个空闲的磁盘块，返回其磁盘块号码。并利用 `map_block(bno)` 函数为这个磁盘块分配一页物理内存，将其加载到内存中，设置权限为 `PTE_V | PTE_R` 。

返回分配好的磁盘块的序号bno。

```
// Overview:
//  Allocate a block -- first find a free block in the bitmap, then map it into
memory.
int alloc_block(void)
{
    int r, bno;
    // Step 1: find a free block.
    if ((r = alloc_block_num()) < 0) { // failed.
        return r;
    }
    bno = r;
    // Step 2: map this block into memory.
    if ((r = map_block(bno)) < 0) {
        free_block(bno);
        return r;
    }
    // Step 3: return block number.
    return bno;
}
```

block_is_free() free_block()在前文总结过了

*Part*2.3 文件系统服务

在之前的章节我们实现了对于在操作系统侧对文件块进行操作，虽然有读写外设和块缓存机制的抽象，但是本质上打开文件之类的操作依然是需要了解磁盘的知识，这显然是不够优雅的，我们可以先看一下用户进程这侧会提供什么，在 `fsipc.c` 中可以看到

```
int fsipc_open(const char *path, u_int omode, struct Fd *fd);
int fsipc_map(u_int fileid, u_int offset, u_int dstva);
int fsipc_set_size(u_int fileid, u_int size);
int fsipc_close(u_int fileid);
int fsipc_dirty(u_int fileid, u_int offset);
int fsipc_remove(const char *path);
```

用户提供的都是 `req_path` 文件路径， `req_fileid` 文件编号， `req_offset` 文件偏移量等一系列参数。不过都可以看出，这些参数都是比较自然的（接近我们对于文件的概念），也就是抽象很好的。这跟磁盘块之类的东西是不相关的，而我们的目的，就是通过 `fs.c, serve.c` 这几个文件，将磁盘块的实现细节给彻底隐藏掉，这样才可以为用户提供更好的服务。

$Part2.3.1$ 传递信息的方式

先来介绍一些宏

```
// Definitions for requests from clients to file system
#define FSREQ_OPEN  1
#define FSREQ_MAP   2
#define FSREQ_SET_SIZE  3
#define FSREQ_CLOSE 4
#define FSREQ_DIRTY 5
#define FSREQ_REMOVE    6
#define FSREQ_SYNC  7

// Max number of open files in the file system at once
#define MAXOPEN         1024
#define FILEVA          0x60000000

// initialize to force into data section
// 打开文件的全局数组
struct Open opentab[MAXOPEN] = { { 0, 0, 1 } };

// Virtual address at which to receive page mappings containing client requests.
// 接受文件系统客户端的请求，并记录 的虚拟页面基地址
#define REQVA   0x0ffff000
```

进程之间想要沟通，就需要传递信息。

`ipc` 提供的传递信息的方式有两种，一种是利用 `env->ipc_value` 进行传值，但是这样只能传递一个 `int` 值，显然是不符合我们的需求的。所以我们只能用共享页面的方式去传递数据。在用户侧，我们会单独分配出固定的一页来进行传递，可以看到

```
// 用来共享的页面
extern u_char fsipcbuf[BY2PG];

// 将页面用不同的形式书写
req = (struct Fsreq_open *)fsipcbuf;
req = (struct Fsreq_map *)fsipcbuf;
...
req = (struct Fsreq_dirty *)fsipcbuf;

// 不同的请求格式
struct Fsreq_open
{
    char req_path[MAXPATHLEN];
    u_int req_omode;
};

struct Fsreq_map
{
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_set_size
{
    int req_fileid;
    u_int req_size;
};

struct Fsreq_close
{
    int req_fileid;
};
```

```
struct Fsreq_dirty
{
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_remove
{
    u_char req_path[MAXPATHLEN];
};
```

这是我们请求服务的方式，那么 `fs` 进程是如何响应服务的呢？基于同样的原因，依然是利用共享页面的方式，我们共享的，正是 `Filefd` 结构体，我们可以看一下其结构

Filefd结构体

```
// file descriptor + file
struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;     // 文件 f_file 的唯一标识
    struct File f_file;
};
```

Fd (file descriptor) 结构体

```
// file descriptor
struct Fd {
    u_int fd_dev_id;    // 文件所处设备的id
    u_int fd_offset;
    u_int fd_omode;     // 文件打开的模式
};
```

File 结构体

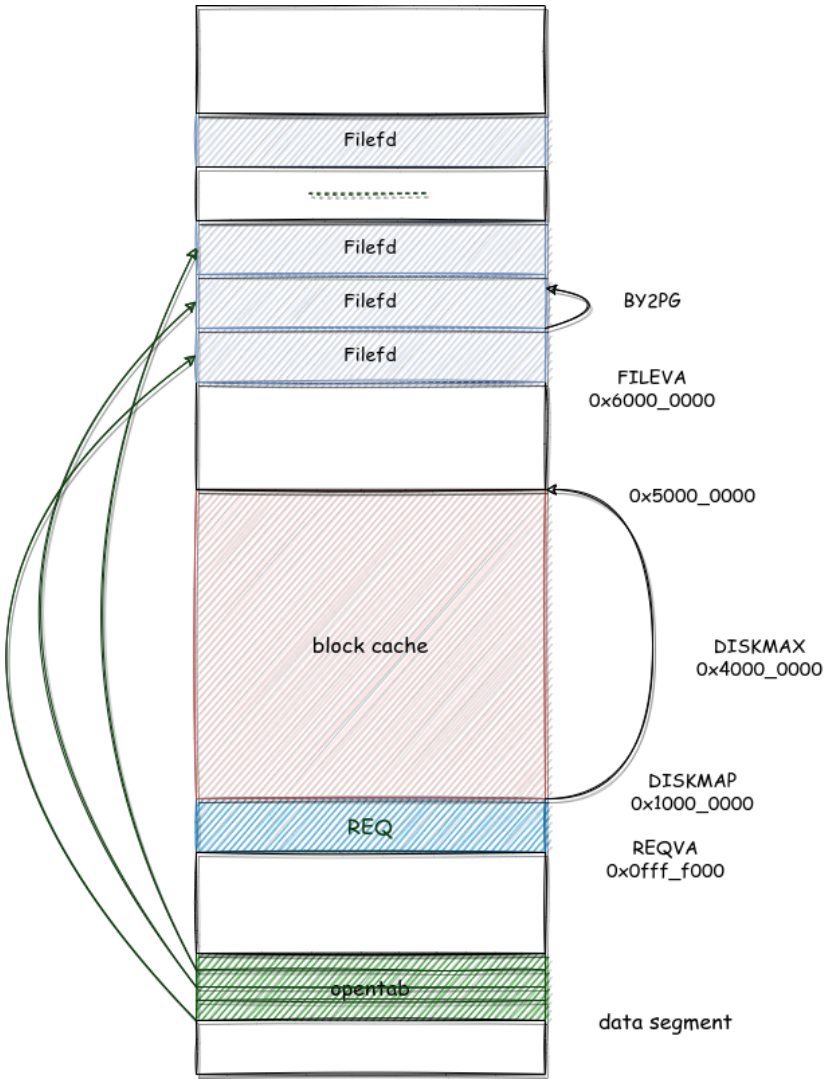
```
struct File
{
    u_char f_name[MAXNAMELEN]; // filename
    u_int f_size;              // file size in bytes
    u_int f_type;              // file type
    u_int f_direct[NDIRECT];
    u_int f_indirect;

    struct File *f_dir; // the pointer to the dir where this file is in, valid
only in memory.
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

可以看到 `Filefd` 里是没有指针的，所以这些信息都是会被完完全全的传递的给用户进程。我们来分析一下， `Fd` 结构体肯定是要传的，因为 `Fd` 是用户侧的抽象， `fileid` 也是要传的，这是文件的唯一标识，不过似乎 `File` 的作用需要限制一下，我们传递 `File` 结构体，是为了使用里面的 `f_name` ， `f_size` ， `f_type` 的信息，而不是为了使用里面记录的 `f_direct, f_indirect, f_dir` 信息，这些信息涉及了磁盘块，对于一般用户进程是不需要知道的。

正是因为采用的是页面共享的方式，所以每个 `Filefd` 必须占据一个页面，不然就会导致一次共享传递了多个 `Fielfd` 。导致了安全问题。

$Part2.3.2$ 服务进程的内存布局

服务进程的内存布局如下：



Open结构体

我们首先来说一下 `struct Open` 结构体

```
struct Open
{
    struct File *o_file; // mapped descriptor for open file
    // 打开文件对应的文件控制块
    u_int o_fileid;      // file id     // 文件唯一标识符
    int o_mode;          // open mode   // 打开方式
    struct Filefd *o_ff; // va of filefd page
    // struct Filefd 所在的虚拟页面的 基地址
};
```

可以看到基本上没有啥重要的信息，因为信息并不是在这里被记录的，而是在这两个指针指向的位置上记录的。第一个指针是一个 `File` 结构体指针，这个指针指向的是类型为 `FILE` 的磁盘块（一个磁盘块上存着 16 个 `File` 结构体），严谨地说，是指向这个磁盘块在内存中的块缓存地址。而 `Filefd` 指针指向的就是盛放 `Filefd` 结构体的页面，是一个顺序映射，可以在初始化函数中看到

```
void serve_init(void)
{
    int i;
```

```
    u_int va;

    // Set virtual address to map.
    va = FILEVA;

    // Initial array opentab.
    for (i = 0; i < MAXOPEN; i++)
    {
        opentab[i].o_fileid = i;
        opentab[i].o_ff = (struct Filefd *)va;
        va += BY2PG;
    }
}
```

$Part2.3.3$ 服务进程的结构

`serve` 进程用一个死循环结合 `switch` 的操作来根据收到的请求不同来提供不同的服务。

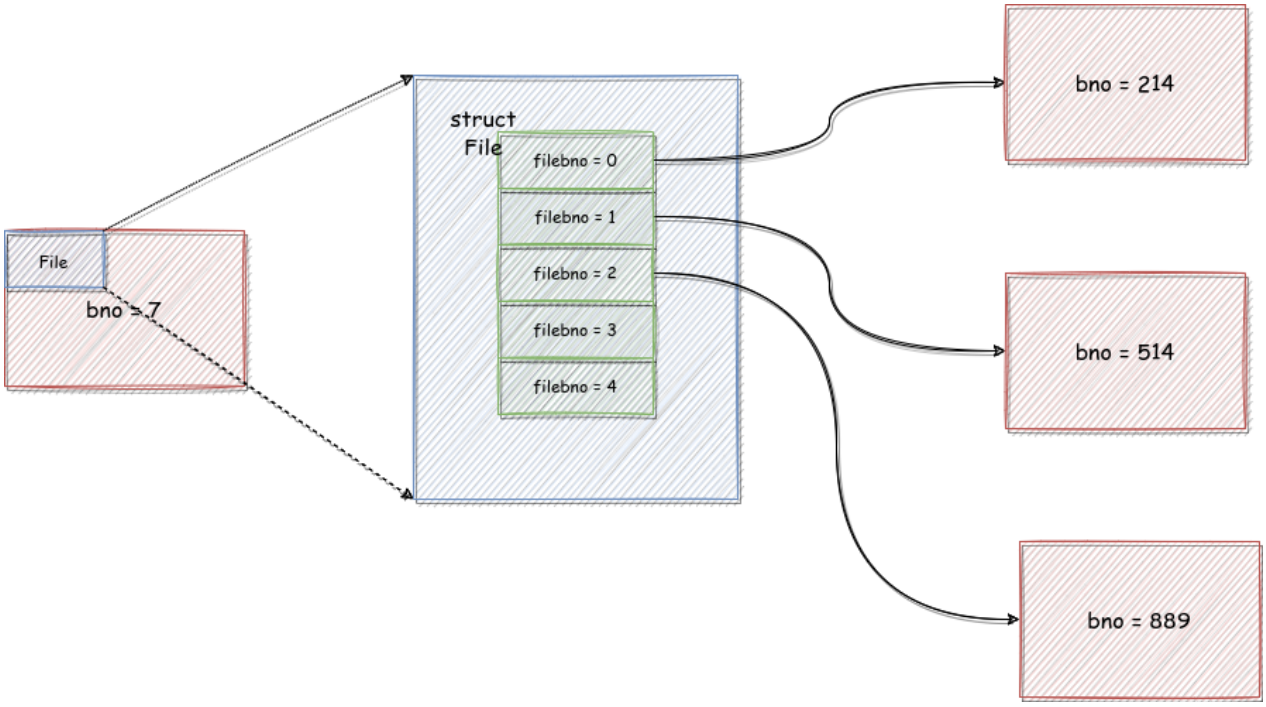
serve(void) 文件系统服务进程

```
void serve(void)
{
    u_int req, whom, perm;

    for (;;)
    {
        perm = 0;
        // 此时，发送信息的进程的envid被记录在了 whom 里
        req = ipc_recv(&whom, REQVA, &perm); // block here

        // All requests must contain an argument page
        if (!(perm & PTE_V))
        {
            writef("Invalid request from %08x: no argument page\n", whom);
            continue; // just leave it hanging, waiting for the next request.
        }

        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:
            writef("Invalid request code %d from %08x\n", whom, req);
            break;
        }
        // 在每次完成服务后，还需要解除映射关系
        syscall_mem_unmap(0, REQVA);
    }
}
```

因为 `ipc_recv` 实现了阻塞机制，所以 `for` 的死循环并不会造成 CPU 的空转。

然后介绍具体的实现方式，我们从下面这张图中可以看出，所谓的查找，就是打开一个个 `File` 结构体，然后根据其记录的指针，去查找磁盘块的过程。



虽然磁盘块的编号 `bno` 是没有规律的（不连续，不单调），但是文件的磁盘块编号是连续一致的，即 `filebno` 。而 `filebno` 是很好与文件的偏移量转换的。所以我们要做的工作就是，在给出偏移量的时候，直接找到对应的磁盘块（一般普通文件都是这样的），在没有给出偏移量的时候，遍历所有的 `filebno` ，找到对应的磁盘块（一般目录是这样的）。

file_block_walk()

> 这个函数是最基础的函数，可以查询出某个文件的某个块.
>
> 他会传入一个文件控制块 `f` ，然后传入需要查询的块 `filebno` ，然后他会有一个参数式的返回值是 `ppdiskno` ，会是磁盘编号，但是我想不清楚为啥是需要二重指针，我觉得一重就够了，然后还有一个 `allco` 负责当查询不到的时候就分配的一个操作。

类似于 `pgdir_walk` 函数。

找到在文件控制块 f 下辖的 第 filebno 个磁盘块的磁盘序号（或者是实验中说的“指针”）

并将参数二重指针 `**ppdiskbno` 指向的值 `*ppdiskbno` 设置为指向这个序号值的一重指针。

> 这个序号是存储在文件控制块 f 的直接指针数组 `f->f_direct[]` 或者是 `f->indirect` 指向的间接指针磁盘块中 的某个值。

当参数 alloc 设置为1时，该函数会在必要的时候为 `f->indirect` 申请一个新的空闲磁盘块，并将其赋值为新申请的磁盘块的序号（在 `f->indirect` 原来是0的情况下）

```
// Overview:
//  Like pgdir_walk but for files.
//  Find the disk block number slot for the 'filebno'th block in file 'f'. Then,
set
//  '*ppdiskbno' to point to that slot. The slot will be one of the f-
>f_direct[] entries, or an entry in the indirect block.
//  When 'alloc' is set, this function will allocate an indirect block if
necessary.
//
// Post-Condition:
//  Return 0: success, and set the pointer to the target block in
*ppdiskbno(Note that the pointer might be NULL).
//  -E_NOT_FOUND if the function needed to allocate an indirect block, but alloc
was 0.
//  -E_NO_DISK if there's no space on the disk for an indirect block.
//  -E_NO_MEM if there's no space in memory for an indirect block.
//  -E_INVAL if filebno is out of range (it's >= NINDIRECT).
int file_block_walk(struct File *f, u_int filebno, u_int **ppdiskbno, u_int
alloc)
{
    int r;
    u_int *ptr;
    void *blk;
    // 如果 filebno 是小于 NDIRECT=10 的，那么在直接指针数组中取出磁盘序号，将其·地址·赋给
ptr即可
    if (filebno < NDIRECT) {
        // Step 1: if the target block is corresponded to a direct pointer, just
return the disk block number.
        ptr = &f->f_direct[filebno];
    } else if (filebno < NINDIRECT) {
        // Step 2: if the target block is corresponded to the indirect block,
but there's no indirect block and `alloc` is set, create the indirect block.
        if (f->f_indirect == 0) {
            if (alloc == 0) {
                return -E_NOT_FOUND;
            }

            if ((r = alloc_block()) < 0) {
                return r;
            }
            f->f_indirect = r;
        }

        // Step 3: read the new indirect block to memory.
        /* 将间接指针磁盘块加载f->f_indirect到物理内存中，并设置 blk 为磁盘块对应的虚拟地
址 */
        if ((r = read_block(f->f_indirect, &blk, 0)) < 0) {
            return r;
        }
        ptr = (u_int *)blk + filebno;/* 设置ptr为指向第filebno个磁盘块序号的值的指针
*/
    } else {
        return -E_INVAL;
    }
    // Step 4: store the result into *ppdiskbno, and return 0.
    *ppdiskbno = ptr;
    return 0;
}
```

file_map_block()

找到在文件控制块 f 下辖的 第 filebno 个磁盘块的磁盘序号，

并将这个序号值作为参数一重指针 `*diskbno` 指向的值。

当 alloc 设置为1的时候，如果要找的磁盘块不存在，新申请一个空闲磁盘块，再令新申请的空闲磁盘块的序号值作为参数一重指针 `*diskbno` 指向的值。

```
// OVerview:
//  Set *diskbno to the disk block number for the filebno'th block in file f.
//  If alloc is set and the block does not exist, allocate it.
//
// Post-Condition:
//  Returns 0: success, < 0 on error.
//  Errors are:
//      -E_NOT_FOUND: alloc was 0 but the block did not exist.
//      -E_NO_DISK: if a block needed to be allocated but the disk is full.
//      -E_NO_MEM: if we're out of memory.
//      -E_INVAL: if filebno is out of range.
int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc)
{
    int r;
    u_int *ptr;

    // Step 1: find the pointer for the target block.
    // 找到在文件控制块 f 下辖的 第 filebno 个磁盘块的磁盘序号，并让ptr指向它
    if ((r = file_block_walk(f, filebno, &ptr, alloc)) < 0) {
        return r;
    }
    // Step 2: if the block not exists, and create is set, alloc one.
    if (*ptr == 0) {
        if (alloc == 0) {
            return -E_NOT_FOUND;
        }
        if ((r = alloc_block()) < 0) {
            return r;
        }
        *ptr = r;
    }
    // Step 3: set the pointer to the block in *diskbno and return 0.
    *diskbno = *ptr;
    return 0;
}
```

file_clear_block()

找到在文件控制块 f 下辖的 第 filebno 个磁盘块，

调用 `free_block()` 函数，利用位图将其状态设置为空闲，

并 **将这个磁盘块 原先存储这个的序号的地方清零**（**这就是为什么** `file_block_walk()` 函数的参数要使用二重指针）。当然，如果本身这个磁盘块就不在内存中，那就成功退出。

```
// Overview:
//  Remove a block from file f.  If it's not there, just silently succeed.
int file_clear_block(struct File *f, u_int filebno)
{
    int r;
    u_int *ptr;
```

```
    if ((r = file_block_walk(f, filebno, &ptr, 0)) < 0) {
        return r;
    }
    if (*ptr) {
        free_block(*ptr);
        *ptr = 0;
    }
    return 0;
}
```

file_get_block()

找到在文件控制块 f 下辖的 第 filebno 个磁盘块，将其读入到内存中，并令 *blk 存储这个磁盘块在**虚拟内存中的地址**。

这个函数就解释了 `file_map_block()` 和 `read_block()` 的不同：

前者只是找到文件控制块 f 下辖的 第 filebno 个磁盘块的序号（没有的话会新申请），存入diskbno，

后者才是真正读取文件控制块 f 下辖的 第 filebno 个磁盘块的内容。读入到内存中。

```
// Overview:
//  Set *blk to point at the filebno'th block in file f.
//
// Hint: use file_map_block and read_block.
//
// Post-Condition:
//  return 0 on success, and read the data to `blk`, return <0 on error.
int file_get_block(struct File *f, u_int filebno, void **blk)
{
    int r;
    u_int diskbno;
    u_int isnew;
    // Step 1: find the disk block number is `f` using `file_map_block`.
    if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) {
        return r;
    }

    // Step 2: read the data in this disk to blk.
    if ((r = read_block(diskbno, blk, &isnew)) < 0) {
        return r;
    }
    return 0;
}
```

*Part*2.3.4 创建一个文件

工具函数 pageref()

得到一个虚拟地址 v 对应的物理内存控制块的pp_ref，也就是物理内存的被引用/映射次数。

```
int pageref(void *v)
{
    u_int pte;
    if (!((* vpd)[PDX(v)]&PTE_V)) {
        return 0;
```

```
    }

    pte = (* vpt)[VPN(v)];
    if (!(pte & PTE_V)) {
        return 0;
    }

    return pages[PPN(pte)].pp_ref;
}
```

open_alloc()

首先看对应的 `Filefd` 有没有空闲，这个理解起来就稍微有一点难度，我们知道本质上 `Filefd` 是一个共享页面，那么这个页面如果正常工作的话，那么应该被引用至少两次，一次是在服务进程处，一次是在打开这个文件的用户进程处。所以如果一个 `Filefd` 的页面被应用次数是 0 或者 1，那么就说明这个页面还没有被分配，那么我们就需要挑选这种 `Filefd` 了。

```
switch (pageref(opentab[i].o_ff))
{
 case 0: // 说明没有被分配
 case 1: // 说明应该被分配了，但是后面有被解除了映射关系，所以只有 fs 进程的一个引用
 default:// 说明已经被分配，就不考虑他们了
}
```

但是判断 `Open` 的占位情况，就骚的不行了。说先说明，他是用 `o_fileid` 这个成员来起“标记脏位的作用的”，但是这个真的恶心，我们知道对于 `opentab` 数组里的 `o_fileid` ，在初始化的时候，他们的值是 `0, 1, 2, 3, ... , 1023` 。只要一位被分配了，我们就让他加上 `1024` ，那么可能就是 `1024, 1025, 3, ..., 1023` 。因为会反复利用，所以有可能出现 `2048, 1025, 1026, ..., 2047` 的情况。

然后就看一下实际功能，其实就是可能需要分配一个页面给 `struct Filefd` ，然后还需要修改一下 `fileid` ，然后清空一下这个页面就完事了。

```
// case 0
syscall_mem_alloc(0, (u_int)opentab[i].o_ff, PTE_V | PTE_R | PTE_LIBRARY));
// case 1
opentab[i].o_fileid += MAXOPEN;
user_bzero((void *)opentab[i].o_ff, BY2PG);
```

只能说对于对 `1024` 取模一致的 `fileid` ，会对应到同一个 `Open` 结构体，进而对应到同一个 `Filefd` 结构体。

```
// Overview:
//  Allocate an open file.
int open_alloc(struct Open **o)
{
    int i, r;
    // Find an available open-file table entry
    for (i = 0; i < MAXOPEN; i++) {
        switch (pageref(opentab[i].o_ff)) {
            case 0:
                if ((r = syscall_mem_alloc(0,
                                           (u_int)opentab[i].o_ff,
                                           PTE_V | PTE_R | PTE_LIBRARY)) < 0) {
                    return r;
                }
            case 1:
                opentab[i].o_fileid += MAXOPEN;
                *o = &opentab[i];
```

```
                user_bzero((void *)opentab[i].o_ff, BY2PG);
                return (*o)->o_fileid;
        }
    }

    return -E_MAX_OPEN;
}
```

serve_open()

这个函数传入进程号和 `Fsreq_open` 请求，打开一个文件。

> 其中的 `o` 是一个 `struct Open` 的结构体，里面的成员 `o_ff` 是一个 `struct Filefd` 结构体。然后就会奇妙的发现，似乎这两个结构体有很多冗余的信息，比如说两者都有与 `struct File` 相关的结构，都有 `fileid` 这类的成员，如果要算上 `Fd` 里面的成员，会发现还有 `omode` 之类的东西，为什么会这么冗余?
>
> 这是因为这两个东西的用处并不一样。从上面的表我们可以得知， `Open` 结构体主要用于记录信息和查表，而 `Filefd` 结构体一个结构体就要占一页，其原因就是他会以共享页面的形式作为服务的反馈传递给一般用户进程。所以从服务进程的角度看，信息确实是冗余了，从下面的代码可以得到例证
>
> ```
> ff = (struct Filefd *)o->o_ff;
> ff->f_file = *f;
> ff->f_fileid = o->o_fileid;
> o->o_mode = rq->req_omode;
> ff->f_fd.fd_omode = o->o_mode;
> ff->f_fd.fd_dev_id = devfile.dev_id;
> ```
>
> 这里进行了一波冗余的赋值，但是从一般用户进程的角度看，并不是冗余的，这是因为一般用户进程只能看到传递的共享页面，即 `struct Filefd` 结构体。

分析这个函数的流程，首先为这个即将打开的文件分配出一个地方，这个地方其实是有两个部分组成的，一个是一个 `struct Open` 的元素（内存映射图中深绿色的条目），一个是 `struct Filefd` 的页面（内存映射图中淡蓝的页面）。

```
void serve_open(u_int envid, struct Fsreq_open *rq)
{
    writef("serve_open %08x %x 0x%x\n", envid, (int)rq->req_path, rq-
>req_omode);

    u_char path[MAXPATHLEN];
    struct File *f;
    struct Filefd *ff;
    int fileid;
    int r;
    struct Open *o;

    // Copy in the path, making sure it's null-terminated
    // 复制请求中要打开的文件路径
    user_bcopy(rq->req_path, path, MAXPATHLEN);
    path[MAXPATHLEN - 1] = 0;

    // Find a file id.
    if ((r = open_alloc(&o)) < 0)
    {
        user_panic("open_alloc failed: %d, invalid path: %s", r, path);
        ipc_send(envid, r, 0, 0);
    }

    fileid = r;

    // Open the file.
```

```
    // 调用函数，打开文件并记录文件控制块的指针
    if ((r = file_open((char *)path, &f)) < 0)
    {
        //  user_panic("file_open failed: %d, invalid path: %s", r, path);
        ipc_send(envid, r, 0, 0);
        return;
    }

    // Save the file pointer.
    o->o_file = f;

    // Fill out the Filefd structure
    ff = (struct Filefd *)o->o_ff;
    ff->f_file = *f;
    ff->f_fileid = o->o_fileid;
    o->o_mode = rq->req_omode;
    ff->f_fd.fd_omode = o->o_mode;
    ff->f_fd.fd_dev_id = devfile.dev_id;

    ipc_send(envid, 0, (u_int)o->o_ff, PTE_V | PTE_R | PTE_LIBRARY);
}
```

file_open()

这个函数实现的功能是给定一个路径，将对应的文件结构体的指针存入参数。

```
// Overview:
//  Open "path".
//
// Post-Condition:
//  On success set *pfile to point at the file and return 0.
//  On error return < 0.
int file_open(char *path, struct File **file)
{
    return walk_path(path, 0, file, 0);
}
```

skip_slash() 跳过字符指针*p中的斜杠 /

```
char * skip_slash(char *p)
{
    while (*p == '/') {
        p++;
    }
    return p;
}
```

walk_path() 极度重要

`walk_path` 这个函数的功能更为强大，给定一个路径，如果找得到这个文件，那么就会返回这个文件的结构体，文件所在目录，如果找不到，那么就会返回文件所在目录和最后一个路径元素

```
int walk_path(char *path, struct File **pdir, struct File **pfile, char
*lastelem)
{
    char *p;
    char name[MAXNAMELEN];
```

```
    struct File *dir, *file;
    int r;

    // start at the root.
    path = skip_slash(path);
    // 让file指向根目录文件控制块
    file = &super->s_root;
    dir = 0;
    name[0] = 0;

    if (pdir) {
        *pdir = 0;
    }
    *pfile = 0;

    // find the target file by name recursively.
    while (*path != '\0') {// 路径未结束
        dir = file;
        p = path;
        while (*path != '/' && *path != '\0') {
            path++;
        }
        if (path - p >= MAXNAMELEN) {// 检查是否超过了单个文件/文件名的长度
            return -E_BAD_PATH;
        }
        // 把这一节路径存入name数组，并令name的结尾是'\0'
        user_bcopy(p, name, path - p);
        name[path - p] = '\0';
        path = skip_slash(path);

        if (dir->f_type != FTYPE_DIR) {// 检查当前文件类型是否为目录
            return -E_NOT_FOUND;
        }

        if ((r = dir_lookup(dir, name, &file)) < 0) {

            // 如果错误是-E_NOT_FOUND 且 已经查询到了末尾
            if (r == -E_NOT_FOUND && *path == '\0') {
                if (pdir) {// 则赋给参数 最后一级目录
                    *pdir = dir;
                }
                if (lastelem) {// 则赋给参数 最后一级 希望找见但没有找见的 文件的名字
                    strcpy(lastelem, name);
                }
                *pfile = 0;
            }
            return r;
        }
    }

    if (pdir) {
        *pdir = dir;
    }
    *pfile = file;
    return 0;
}
```

dir_lookup()

查找某个目录 dir 下是否存在名字叫 name 的文件。如果找到了，就将指向该文件的指针存放在参数 *file 中。

file_get_block可以将某个指定文件指向的磁盘块读入内存。

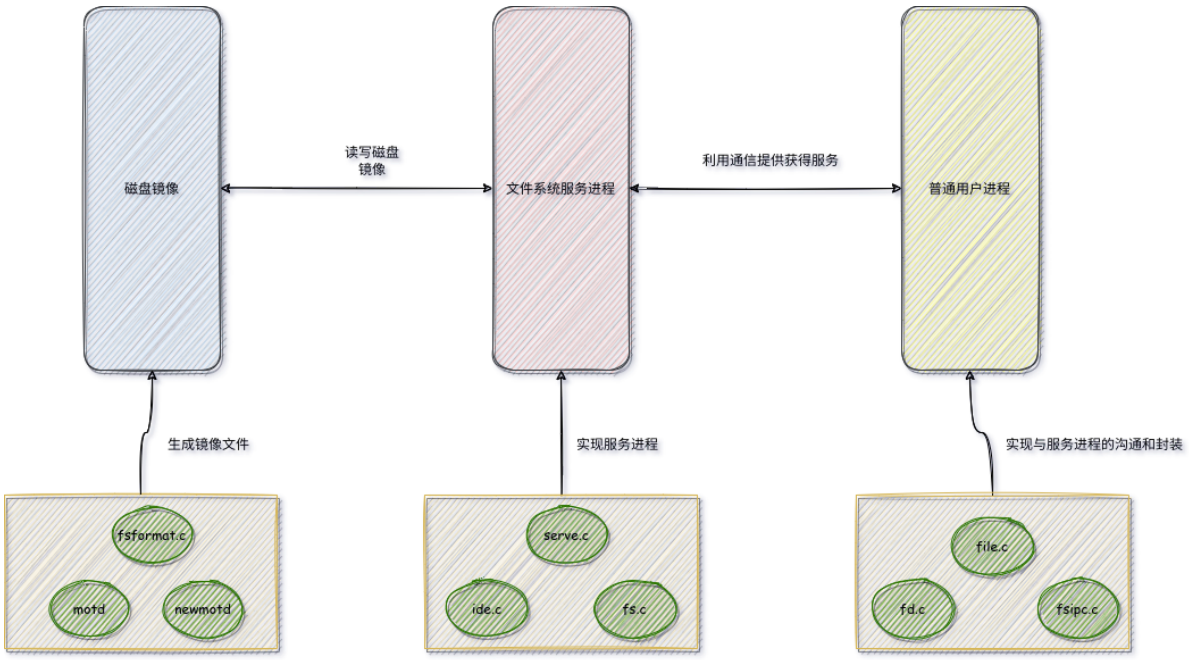
`dir_lookup` 其实现方法就是：找到这个目录内容对应的所有磁盘块（第一重循环，遍历 `filebno`，利用 `file_get_block` 获得磁盘块对应的地址 `blk`），然后遍历每个磁盘块中的每个文件结构体 `File` （第二重循环）。然后对比文件名，获得结果。

```
// Overview:
//  Try to find a file named "name" in dir.  If so, set *file to it.
//
// Post-Condition:
//  return 0 on success, and set the pointer to the target file in `*file`.
//      < 0 on error.
int dir_lookup(struct File *dir, char *name, struct File **file)
{
    int r;
    u_int i, j, nblock;
    void *blk;
    struct File *f;

    // Step 1: Calculate nblock: how many blocks are there in this dir?
    nblock = ROUND(dir->f_size, BY2BLK) / BY2BLK;/* 计算当前目录管辖的文件总共占用多少磁盘块 */

    for (i = 0; i < nblock; i++) {
        // Step 2: Read the i'th block of the dir.
        // Hint: Use file_get_block.
        r = file_get_block(dir, i, &blk);/* 将当前目录下第 i 个磁盘块的内容加载到内存中，并且设置blk为其在虚拟内存中的地址 */
        if (r < 0) {
            return r;
        }
        // Step 3: Find target file by file name in all files on this block.
        // If we find the target file, set the result to *file and set f_dir field.
        for (j = 0; j < FILE2BLK; j++) {
            f = ((struct File *)blk) + j;/* f 为第 i 个磁盘块中第 j 个文件控制块的指针 */
            if (strcmp((char *)f->f_name, name) == 0) {
                f->f_dir = dir; /* 设置其目录为当前目录dir */
                *file = f; /* 让参数为该文件控制块的指针 */
                return 0;
            }
        }
    }
    return -E_NOT_FOUND;
}
```

$Part2.4$ 文件系统的用户接口

文件系统在建立之后，需要向用户提供相关的接口使用。我们实验使用的操作系统内核符合一个典型的微内核的设计，文件系统属于用户态进程，以服务的形式供其他进程调用。这个过程中，不仅涉及了不同进程之间通信的问题，也涉及了文件系统如何隔离底层的文件系统实现，抽象地表示一个文件的问题。我们引入文件描述符（file descriptor）作为用户程序管理、操作文件资源的方式。

如何理解文件系统的微内核？这里摘录强生大佬的博客

> 这一章的微内核与 lab4 的异常处理的微内核是两个东西，这是最重要的一点。
>
> 微内核设计都强调将原本属于操作系统的一部分功能分配出来（比如说异常处理，文件系统），分配给用户进程完成。在 lab4 中，微内核设计是把异常处理的功能分出来，让每个进程都具有异常处理的功能（以库函数的形式）。
>
> 但是在 lab5 中，微内核设计确实将文件系统的实现分了出来，但是**并不是配置到了每一个用户进程上**，让每一个进程都有文件系统的能力，而是**专门开设了一个用户进程专门提供文件系统的服务**（这也是这个用户进程的大部分实现文件都被放置在了 `fs` 文件夹下，而不是 `user` 文件夹下的原因），而其他进程想要获得文件系统的服务，就要与这个特殊的用户进程进行通信，就好像之前与操作系统进行通信一样。
>
> 
>

下面再介绍一些结构体和宏。

```
#define debug 0

#define MAXFD 32                        // 最多的文件描述符的个数
#define FILEBASE 0x60000000             // 这里是 文件系统基地址
#define FDTABLE (FILEBASE-PDMAP)        // 文件描述符数组的基地址，腾出了一个PDMAP的空间，
也就是4M
#define INDEX2FD(i) (FDTABLE+(i)*BY2PG)// 一个文件描述符占据一页大小
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)// 一个文件块占据一个PDMAP大小

// State
struct Stat {
    char st_name[MAXNAMELEN];
    u_int st_size;
    u_int st_isdir;
    struct Dev *st_dev;
};
```

外设不只是磁盘，可能还有控制台和管道，总之这些东西的读写都是不一样的，如果每个东西都实现一遍，显然是有些浪费和冗余的。所以最好的就是我们抽象出一个对于文件、控制台、管道都适用的部分，这个部分用一个统一的东西来描述，而对于存在差异的部分，再分别实现，而这种结构可以用函数指针很好的实现。

```
// Device struct:
// It is used to read and write data from corresponding device.
// We can use the five functions to handle data.
// There are three devices in this OS: file, console and pipe.
struct Dev {
    int dev_id;
    char *dev_name;
    int (*dev_read)(struct Fd *, void *, u_int, u_int);
    int (*dev_write)(struct Fd *, const void *, u_int, u_int);
    int (*dev_close)(struct Fd *);
    int (*dev_stat)(struct Fd *, struct Stat *);
    int (*dev_seek)(struct Fd *, u_int);
};
```

可以看到，这个结构体里面定义了 `id` ， `name` ，最关键的是定义了一大堆的函数指针，包括读、写、关闭、定位等函数。只需要将这些函数指针在用到的时候替换成合适的函数，就可以实现上述目的，比如说：

```
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,
};
```

其中 `file_read, file_write, file_close` 等函数都是具体的存在特异性的实现，它们是在 `file.c` 中具体实现的。

这里插一嘴闲话，这个部分越来越像面向对象的感觉了，函数指针类似于多态的实现，而各种指针的转换很像向下类型转换。

最后梳理一下这个部分的内容。对于所有的外设，我们统一抽象出一个叫做“文件描述符” `fd` 的东西去管理（不只是文件，或者说这正是“一切皆文件”哲学的体现）。但是依然是需要具体实现的，对于文件而言，我们把读写之类的具体函数放到了 `file.c` 文件中实现。

文件描述符

Fd (file descriptor) 结构体

```
// file descriptor
struct Fd {
    u_int fd_dev_id;    // 文件所处设备的id
    u_int fd_offset;
    u_int fd_omode;     // 文件打开的模式
};
```

> `fd` 中记录着我们打开文件的时候的状态信息，我们平时也不用 `fd` ，而是用一个索引去查找对应的 `fd` 。这个索引被成为 `fdnum` ，我们一般都会用这个东西。而 `fd` 会规规矩矩的存在一个地方（就像一个数组一样）。

> 那么为什么不直接做成一个 `fd` 的数组呢？在之前说为说过如果有共享页面的需求的话，可以一个数组元素单独存放一页，但是 `Fd` 数组是没有这个需求的，那么为啥不直接开一个数组。我觉得是这个原因，是因为最顶层用户是没有 `Filefd` 结构体的，用户只能看见 `Fd` 结构体，相当于 `File` 结构体被隐藏了。所以为了少让用户看见，干脆一个 `Fd` 开一页，这样也方便将 `Fd` 转换成 `Filefd` 。

当用户进程试图打开一个文件时，需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。当用户进程向文件系统发送打开文件的请求时，文件系统进程会将这些基本信息记录在内存中，然后由操作系统将用户进程请求的地址映射到同一个物理页上，因此一个文件描述符至少需要独占一页的空间。当用户进程获取了文件大小等基本信息后，再次向文件系统发送请求将文件内容映射到指定内存空间中。

当我们要读取一个大文件中间的一小部分内容时，从头读到尾是极为浪费的，因此我们需要一个指针帮助我们在文件中定位，在 C 语言中拥有类似功能的函数是fseek。而在读写期间，每次读写也会更新该指针的值。

```
extern u_char fsipcbuf[BY2PG];      // page-aligned, declared in entry.S
    .globl fsipcbuf
fsipcbuf:
    .space BY2PG
// Definitions for requests from clients to file system
#define FSREQ_OPEN  1
#define FSREQ_MAP   2
#define FSREQ_SET_SIZE  3
#define FSREQ_CLOSE 4
#define FSREQ_DIRTY 5
#define FSREQ_REMOVE    6
#define FSREQ_SYNC  7

struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
};

struct Fsreq_map {
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_set_size {
    int req_fileid;
    u_int req_size;
};

struct Fsreq_close {
    int req_fileid;
};

struct Fsreq_dirty {
    int req_fileid;
    u_int req_offset;
};

struct Fsreq_remove {
    u_char req_path[MAXPATHLEN];
};
```

fd_alloc()

从 0 到 MAXFD-1 找到最小的 i，满足：第i个文件描述符结构体没有映射到对应的虚拟页面——文件描述符页面上，并设置参数 *fd 成为指向这个文件描述符结构体的指针。

（不要分配页面。由调用函数自己来分配页面。这意味着，如果有函数连续两次调用fd_ alloc()，而没有分配我们返回的第一个页面，那么我们将在第二次返回相同的页面。）

```
int fd_alloc(struct Fd **fd)
{
    /* 从 0 到 MAXFD-1 找到最小的i，满足：第i个文件描述符结构体没有映射到对应的虚拟页面—文件描述符页面上，并设置参数 *fd 成为指向这个文件描述符结构体的指针。 */
    // Find the smallest i from 0 to MAXFD-1 that doesn't have
    // its fd page mapped.  Set *fd to the fd page virtual address.
    // (Do not allocate a page.  It is up to the caller to allocate
    // the page.  This means that if someone calls fd_alloc twice
    // in a row without allocating the first page we return, we'll
    // return the same page the second time.)
    // Return 0 on success, or an error code on error.
    u_int va;
    u_int fdno;

    for (fdno = 0; fdno < MAXFD - 1; fdno++) {
        va = INDEX2FD(fdno);// va 求出第i个文件描述符结构体的地址
        /* 检验 va 对应一级页表项 的有效性 */
        if (((* vpd)[va / PDMAP] & PTE_V) == 0) {
        // 这里直接令参数 文件描述结构体 fd 成为指向 va 的指针
            *fd = (struct Fd *)va;
            return 0;
        }
        /* 检验 va 对应的二级页表项 的有效性 */
        if (((* vpt)[va / BY2PG] & PTE_V) == 0) {   //the fd is not used
        // 这里直接令参数 文件描述结构体 fd 成为指向 va 的指针
            *fd = (struct Fd *)va;
            return 0;
        }
    }
    return -E_MAX_OPEN;
}
```

fsipc()

向文件服务器发送一个IPC请求，并等待回复。

- type，请求类型，如 `FSREQ_OPEN`
- fsreq，用各自请求结构体方式定义步长后的 fsipcbuf，如 `(struct Fsreq_open *)fsipcbuf`
- dstva，实际上是文件描述符 struct Fd *fd
- perm，权限

```
// Overview:
//  Send an IPC request to the file server, and wait for a reply.
//
// Parameters:
//  @type: request code, passed as the simple integer IPC value.
//  @fsreq: page to send containing additional request data, usually fsipcbuf.
//      Can be modified by server to return additional response info.
//  @dstva: virtual address at which to receive reply page, 0 if none.
//  @*perm: permissions of received page.
//
// Returns:
//  0 if successful,
//  < 0 on failure.
```

```
static int fsipc(u_int type, void *fsreq, u_int dstva, u_int *perm)
{
    u_int whom;
    // NOTEICE: Our file system no.1 process!
    /* 因为第一个进程被拿出来作为文件系统的进程，所以用户进程需要向这个进程发送请求
        就是发送一个请求类型
    */
    ipc_send(envs[1].env_id, type, (u_int)fsreq, PTE_V | PTE_R);
    return ipc_recv(&whom, dstva, perm);
    // 用于接受一个value值，并将接收到的value作为函数返回值，并将信息发送者的envid存储在
*whom中；将此次信息转递的信息权限存储在*perm中
}
```

fsipc_open()

向文件服务器发送 打开文件请求(file-open request)。请求中包含文件路径path和打开模式omode。并根据回复(reply)设置*fileid 和 *size 。

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

    req = (struct Fsreq_open *)fsipcbuf;

    // The path is too long.
    if (strlen(path) >= MAXPATHLEN) {
        return -E_BAD_PATH;
    }

    strcpy((char *)req->req_path, path);
    req->req_omode = omode;
    return fsipc(FSREQ_OPEN, req, (u_int)fd, &perm);
}
```

fsipc_map()

```
int fsipc_map(u_int fileid, u_int offset, u_int dstva)
{
    int r;
    u_int perm;
    struct Fsreq_map *req;

    req = (struct Fsreq_map *)fsipcbuf;
    req->req_fileid = fileid;
    req->req_offset = offset;

    if ((r = fsipc(FSREQ_MAP, req, dstva, &perm)) < 0) {
        return r;
    }

    if ((perm & ~(PTE_R | PTE_LIBRARY)) != (PTE_V)) {
        user_panic("fsipc_map: unexpected permissions %08x for dstva %08x",
perm, dstva);
    }
    return 0;
}
```

Open()

以模式 mode 打开指定路径 path 下 的文件。

```
u_int fd2data(struct Fd *fd)
{
    return INDEX2DATA(fd2num(fd));
}
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)
#define FILEBASE 0x60000000
#define INDEX2FD(i) (FDTABLE+(i)*BY2PG)
#define FDTABLE (FILEBASE-PDMAP)

int open(const char *path, int mode)
{
    struct Fd *fd;
    struct Filefd *ffd;
    u_int size, fileid;
    int r;
    u_int va;
    u_int i;

    // Step 1: Alloc a new Fd, return error code when fail to alloc.
    // Hint: Please use fd_alloc.
    r = fd_alloc(&fd); // this is a whole 4KB space
    if (r < 0) {
        return r;
    }

    // Step 2: Get the file descriptor of the file to open.
    // Hint: Read fsipc.c, and choose a function.
    r = fsipc_open(path, mode, fd);
    if (r < 0) {
        return r;
    }

    // Step 3: Set the start address storing the file's content. Set size and
fileid correctly.
    // Hint: Use fd2data to get the start address.
    ffd = (struct Filefd *)fd;
    va = fd2data(fd);
    fileid = ffd->f_fileid;
    size = ffd->f_file.f_size;

    // Step 4: Alloc memory, map the file content into memory.
    // 以一页4K为大小
    for (i = 0; i < size; i += BY2BLK) {
        r = fsipc_map(fileid, i, va + i);
        if (r < 0) {
            return r;
        }
    }

    // Step 5: Return the number of file descriptor.
    return fd2num(fd);
}
```

read()

```
// Overview:
//  Read 'n' bytes from 'fd' at the current seek position into 'buf'.
//
// Post-Condition:
//  Update seek position.
//  Return the number of bytes read successfully.
//      < 0 on error
int read(int fdnum, void *buf, u_int n)
{
    int r;
    struct Dev *    dev;
    struct Fd *fd;

    // Similar to 'write' function.
    // Step 1: Get fd and dev.
    if ((r = fd_lookup(fdnum, &fd)) < 0
        || (r = dev_lookup(fd->fd_dev_id, &dev)) < 0) {
        return r;
    }
    // Step 2: Check open mode.
    if ((fd->fd_omode & O_ACCMODE) == O_WRONLY) {
        writef("[%08x] read %d -- bad mode\n", env->env_id, fdnum);
        return -E_INVAL;
    }
    if (debug)
        writef("read %d %p %d via dev %s\n",
               fdnum, buf, n, dev->dev_name);
    // Step 3: Read starting from seek position.
    r = (*dev->dev_read)(fd, buf, n, fd->fd_offset);
    // Step 4: Update seek position and set '\0' at the end of buf.
    if (r > 0) {
        fd->fd_offset += r; // 读取偏移量加上r
    }
    ((char *)buf)[r] = '\0';

    return r;
}
```

write()

```
int write(int fdnum, const void *buf, u_int n)
{
    int r;
    struct Dev *dev;
    struct Fd *fd;

    if ((r = fd_lookup(fdnum, &fd)) < 0
        ||  (r = dev_lookup(fd->fd_dev_id, &dev)) < 0) {
        return r;
    }

    if ((fd->fd_omode & O_ACCMODE) == O_RDONLY) {
        writef("[%08x] write %d -- bad mode\n", env->env_id, fdnum);
        return -E_INVAL;
    }
```

```
    if (debug) writef("write %d %p %d via dev %s\n",
                          fdnum, buf, n, dev->dev_name);

    r = (*dev->dev_write)(fd, buf, n, fd->fd_offset);

    if (r > 0) {
        fd->fd_offset += r;
    }

    return r;
}
```

文件系统服务

Bonus 结合代码来理解：文件系统(file system, fs)初始化

一些fs.h中的宏的解读

```
/* IDE disk number to look on for our file system */
#define DISKNO      1

#define BY2SECT     512 /* Bytes per disk sector */
// 一个磁盘块拥有的扇区数目
#define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP     0x10000000  // 磁盘映射到内存中的基地址

/* Maximum disk size we can handle (1GB) */
#define DISKMAX     0x40000000  // 磁盘的最大尺寸：1G
```

重要的全局变量

```
// 相当于超级磁盘块的指针
struct Super *super;
// 相当于位图磁盘块的指针
u_int *bitmap;      // bitmap blocks mapped in memory
```

fs_init()

从这个函数我们就可以看出，文件系统的初始化实际上是分为三步：

1. 读超级块（read super block），调用 `read_super()`
2. 检查整个磁盘是否可以工作（check if the disk can work），调用 `check_write_block()`
3. 将记录整个磁盘的磁盘块是否空闲的位图磁盘块由磁盘读入内存（read bitmap blocks from disk to memory），调用 `read_bitmap()`

```
// Overview:
//  Initialize the file system.
void fs_init(void)
{
    read_super();
    check_write_block();
    read_bitmap();
}
```

在解读这三个函数之前，我们需要先理解一些通用且重要的工具函数。

文件系统中重要的三个工具函数：diskaddr()、read_block()、write_block()

diskaddr()

fs/fs.c 中的 diskaddr 函数用来计算指定磁盘块对应的虚存地址。

根据一个块的序号bno (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。

核心公式：$\mathrm{DISKMAP} + \mathrm{blockno} * \mathrm{BY2BLK};$

```
// Overview:
//  Return the virtual address of this disk block. If the `blockno` is greater
//  than disk's nblocks, panic.
u_int diskaddr(u_int blockno)
{
    if (super && blockno >= super->s_nblocks) {// 这里的super->s_nblocks就是NBLOCK
= 1024
        user_panic("reading non-existent block %08x\n", blockno);
    }
    return DISKMAP + blockno * BY2BLK;
}
```

read_block()

将指定编号的磁盘块读入到内存中。

将传入的参数 blockno 代表的磁盘块读入到内存中。首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 ide_read 函数来读取磁盘上的数据到对应的虚存地址处。

- 如果说参数blk != 0，就设置参数 *blk 为 blockno 代表的磁盘块在**虚拟内存中的地址**。
- 如果说参数isnew != 0
    - 并且 blockno 代表的磁盘块早就已经读入到了物理内存和进程虚拟空间中，就设置isnew = 0；
    - 并且 blockno 代表的磁盘块还没有读入到物理内存和进程虚拟空间中，就设置isnew = 1。然后为当前进程的进程空间下的该磁盘块对应的虚拟地址 `va = diskaddr(blockno)` 申请一页实际的物理内存用于保存磁盘块上的数据，再将该磁盘块上的数据加载到物理内存和进程虚拟空间；

根据该函数的oveview，这个函数一个重要的作用是确保某个特定的磁盘块已经被加载到了物理内存和用户进程空间。

```
// Overview:
//  Make sure a particular disk block is loaded into memory.
//
// Post-Condition:
//  Return 0 on success, or a negative error code on error.
//
//  If blk!=0, set *blk to the address of the block in memory.
//
//  If isnew!=0, set *isnew to 0 if the block was already in memory, or
//  to 1 if the block was loaded off disk to satisfy this request. (Isnew
//  lets callers like file_get_block clear any memory-only fields
//  from the disk blocks when they come in off disk.)
//
// Hint:
//  use diskaddr, block_is_mapped, syscall_mem_alloc, and ide_read.
int read_block(u_int blockno, void **blk, u_int *isnew)
{
    u_int va;
    // Step 1: validate blockno. Make file the block to read is within the disk.
    /* 检验 blockno 的有效性，如果super块指针不为空，且blockno超过了总共磁盘块数目，那么就
说明读到了不存在的磁盘块，报错 */
    if (super && blockno >= super->s_nblocks) {
        user_panic("reading non-existent block %08x\n", blockno);
    }
    // Step 2: validate this block is used, not free.
    /* 检验 blockno 代表的磁盘块是否是空闲的，如果是，则报错（因为我们要读出一个已经使用了的
磁盘块）
        ! 这里值得注意的是:
```

```
        每当我们使用 block_is_free(blockno) 函数去检查 blockno 代表的磁盘块是否是空闲磁盘块的时候，我们必须确保全局变量 bitmap!=NULL，即我们已经把 位图磁盘块 从磁盘中读入了内存。
        bitmap就是指向 位图磁盘块的指针
    */
    if (bitmap && block_is_free(blockno)) {
        user_panic("reading free block %08x\n", blockno);
    }
    // Step 3: transform block number to corresponding virtual address.
    // 根据 blockno 计算出 该磁盘块应该被加载到的 512B内存虚拟空间的 基地址
    va = diskaddr(blockno);
    // Step 4: read disk and set *isnew.
    // 如果 blockno 对应的磁盘块已经映射到了内存（即加载到了内存中）
    // 那么只在 isnew 不为0时，设置 *isnew 为0（说明不是新的磁盘块）
    // 否则就在 isnew 不为0时，设置 *isnew 为1（说明是新的磁盘块）
    // 并且为新的磁盘块申请一页物理内存（PTE_V | PTE_R），并将磁盘块读到 内存中 上一步计算出的va处
    // 我们只有一个磁盘，所以 使用 ide_read 函数时，磁盘号应该是 0
    // Hint: if this block is already mapped, just set *isnew, else alloc memory and
    // read data from IDE disk (use `syscall_mem_alloc` and `ide_read`).
    // We have only one IDE disk, so the diskno of ide_read should be 0.
    if (block_is_mapped(blockno))
    { //the block is in memory
        if (isnew)
            *isnew = 0;
    }
    else
    { //the block is not in memory
        if (isnew)
            *isnew = 1;
        syscall_mem_alloc(0, va, PTE_V | PTE_R);
        /* 为当前进程的进程空间下的该磁盘块对应的虚拟地址 va = diskaddr(blockno)
           申请一页实际的物理内存用于保存磁盘块上的数据，
           并将这页物理内存的物理页号+权限（PTE_V | PTE_R）存放在
           当前进程页目录映射的二级页表体系中对应的二级页表项中 */
        ide_read(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
        /* 从0号磁盘的第 blockno * SECT2BLK 个扇区开始，将 SECT2BLK 个扇区（也就是一整个磁盘块）
           的磁盘内容读入到 虚拟内存空间 的 va 处*/
    }
    // Step 5: if blk != NULL, set `va` to *blk.
    if (blk)
        *blk = (void *)va;
    return 0;
}
```

write_block()

将序号为 blockno 的磁盘块 disk[blockno] 对应的**虚存空间内中整个磁盘块的数据** 写入到 **磁盘块** 中。

```
void write_block(u_int blockno)
{
    u_int va;
    // Step 1: detect is this block is mapped, if not, can't write it's data to disk.
    // 检查序号为 blockno 的磁盘块是否已经加载到了物理内存
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }
    // Step2: write data to IDE disk. (using ide_write, and the diskno is 0)
    va = diskaddr(blockno);
    // 选择0号磁盘，将 va 代表的地址开始 SECT2BLK 个扇区（也就是一整个磁盘块）的数据，写入到
```

```
    // 0号磁盘的 从第 blocknoo * SECT2BLK个扇区开始 的 SECT2BLK(8) 个扇区中
    ide_write(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
    // 这里由于同一个进程的同一个地址，所以这里的syscall_mem_map调用的page_insert其实只是
    // 更新tlb 和 更新pp对应物理页面基地址的权限（因为在lab4里面会给父进程的页表项新增
PTE_COW用于保护，lab5里面会新增PTE_LIBRARY）
    syscall_mem_map(0, va, 0, va, (PTE_V | PTE_R | PTE_LIBRARY));
}
```

那么，下面我们来看一下文件系统初始化过程中调用的三个函数

read_super() 将磁盘中的超级块读入内存并检查

```
// Overview:
//  Read and validate the file system super-block.
//
// Post-condition:
//  If error occurred during read super block or validate failed, panic.
void read_super(void)
{
    int r;
    void *blk;
    // Step 1: read super block.
    // 读取超级磁盘块（即1号磁盘块）到内存中，并将加载到的内存地址存入blk，再存入super
    if ((r = read_block(1, &blk, 0)) < 0) {
        user_panic("cannot read superblock: %e", r);
    }
    super = blk;

    // Step 2: Check fs magic nunber.    核对 magic nunber
    if (super->s_magic != FS_MAGIC) {
        user_panic("bad file system magic number %x %x", super->s_magic,
FS_MAGIC);
    }

    // Step 3: validate disk size.
    // 检查当前文件系统占用的磁盘块数目是否超过了磁盘的最多磁盘块 1G / 4K = 1024*1024/4 =
256 K
    if (super->s_nblocks > DISKMAX / BY2BLK) {
        user_panic("file system is too large");
    }

    writef("superblock is good\n");
}
```

check_write_block() 检查write_block()函数能否正常工作，实际上读入了启动磁盘块

```
// Overview:
//  Test that write_block works, by smashing the superblock and reading it back.
void check_write_block(void)
{
    super = 0;

    // backup the super block.
    // copy the data in super block to the first block on the disk.
    read_block(0, 0, 0); // 将启动磁盘块（即0号磁盘块）读入内存
    // 用 虚拟内存中 1 号磁盘块中的内容 覆盖 刚才读入的 虚拟内存中 0 号磁盘块中的内容
    // 即用 0 号磁盘块对应的虚拟空间 去 备份一份 1 号磁盘块的数据，这里调用一次read_block是
以防
    // 0号磁盘块还没有申请对应的物理空间和虚拟空间。
    user_bcopy((char *)diskaddr(1), (char *)diskaddr(0), BY2PG);
```

```
    // smash it
    // 这里也是利用s_magic成员是struct Super结构体中的第一个成员，所以可以用基地址直接找到并覆盖
    strcpy((char *)diskaddr(1), "OOPS!\n");
    // 将内存中 1 号磁盘块 对应的虚拟空间的数据 写入磁盘的 1 号磁盘块区域
    write_block(1);
    user_assert(block_is_mapped(1));

    // clear it out
    // 解除原先建立的 1 号磁盘块的虚拟地址和物理内存的映射关系
    // 将进程页目录的二级页表体系中 va=diskaddr(1) 对应的二级页表项清零，顺便减少物理页面引用次数，看情况释放物理页面。做完这些后更新TLB。
    syscall_mem_unmap(0, diskaddr(1));
    user_assert(!block_is_mapped(1));

    // validate the data read from the disk.
    read_block(1, 0, 0); // 将超级磁盘块（即1号磁盘块）读入内存 （实际上是再次读入）
    // 比较刚才 先是在 1号磁盘对应虚拟内存 中写入的“OOPS”
    // 是否真的被写入了实际磁盘1号磁盘块区域
    user_assert(strcmp((char *)diskaddr(1), "OOPS!\n") == 0);

    // restore the super block.
    // 用 0 号磁盘块对应的虚拟空间 备份的 1 号磁盘块的数据还原磁盘 1 号磁盘块的数据
    user_bcopy((char *)diskaddr(0), (char *)diskaddr(1), BY2PG);
    write_block(1);
    // IMPORTANT 在这里，全局变量 super 正式被赋值为 虚拟内存中超级磁盘块（1号）的指针
    super = (struct Super *)diskaddr(1);
}
```

read_bitmap() 将所有位图磁盘块读入内存

```
// Overview:
//  Read and validate the file system bitmap.
//
// Hint:
//  Read all the bitmap blocks into memory.
//  Set the "bitmap" pointer to point ablocknot the beginning of the first bitmap block.
//  For each block i, user_assert(!block_is_free(i))).Check that they're all marked as inuse
void read_bitmap(void)
{
    u_int i;
    void *blk = NULL;

    // Step 1: calculate this number of bitmap blocks, and read all bitmap blocks to memory.
    // 计算出当前文件系统占用的磁盘块个数 需要多少个 位图磁盘块
    nbitmap = super->s_nblocks / BIT2BLK + 1;
    // 将这些位图磁盘块 读入内存
    for (i = 0; i < nbitmap; i++) {
        read_block(i + 2, blk, 0);
    }
    // IMPORTANT 在这里，全局变量 bitmap 正式被赋值为 虚拟内存中位图磁盘块（2号）的指针
    bitmap = (u_int *)diskaddr(2);

    // Step 2: Make sure the reserved and root blocks are marked in-use.
    // Hint: use `block_is_free`
    // 利用读入内存的 位图磁盘块 检查 0号 1号 磁盘块是否是已经使用的状态
    user_assert(!block_is_free(0));
    user_assert(!block_is_free(1));

    // Step 3: Make sure all bitmap blocks are marked in-use.
    // 利用读入内存的 位图磁盘块 检查 所有的位图磁盘块是否是已经使用的状态
```

```
    for (i = 0; i < nbitmap; i++) {
        user_assert(!block_is_free(i + 2));
    }

    writef("read_bitmap is good\n");
}
```

$Part3$ 体会与感想&指导书反馈

感谢BUAA OS Lab5-2分析 - Booooomb - 博客园 (cnblogs.com)

这次这篇博客的图极大地在宏观理解文件系统上帮助了我，真的很感谢。