文件系统

Lab5 中的主要内容

1. 硬件外设与读写驱动
2. 磁盘结构与驱动
3. 文件系统服务进程

衍生操作

```
// 假设目录 dirf 已知，我们实现对它的操作
struct File *dirf;

// 某个目录 dir(File *) 下的磁盘块数量
int nblk = dirf->f_size / BLOCK_SIZE; // (BLOCK_SIZE = 4KB)

// 由 blockno 得到对应磁盘块的文件控制块
struct File *blk = (struct File *)disk[bno].data;

// 由磁盘块缓存地址 blk，遍历该磁盘块上的所有文件控制块
for (struct File *f = blk; f < blk + FILE2BLK; f++) {
  // do sth to f...
}

// 根据某磁盘的文件控制块，判断文件是否为空
if (f->f_name[0] == '\0') {
  // is empty
}

// 申请一个新的磁盘块，通过 map_link_block 获取当前目录文件 dirf 的第 nblk 个数据块的 blockno，并将该 block 的数据部分解释为一个 struct File*
struct File *blk = (struct File *)(disk[make_link_block(dirf, nblk)].data);
```

用户读写磁盘

按物理地址读写

这种读写方式不仅适用于磁盘，实际上适用于对所有外设 / 硬件设备的访问，此时用户需要给出设备的物理地址 pa 来进行访问

IDE 磁盘驱动程序位于用户空间，但是用户进程无法直接写内核虚拟地址，因此需要使用系统调用 syscall_read_dev 和 syscall_write_dev 来实现

```
int syscall_write_dev(void *va, u_int dev, u_int len);
//user/lib/syscall_lib.c
int sys_write_dev(u_int va, u_int pa, u_int len); //kern/syscall_all.c
int syscall_read_dev(void *va, u_int dev, u_int len);
//user/lib/syscall_lib.c
int sys_read_dev(u_int va, u_int pa, u_int len); //kern/syscall_all.c
```

注意，dev 和 pa 是硬件设备在磁盘中的**物理地址**

按磁盘块号读写

这种读写方式专门用于本次 lab 实验中对磁盘块（文件）的访问，此时用户通过给出的磁盘块号 diskno 进行访问

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);
//diskno: 待操作的磁盘编号
//secno: 第一个待操作的扇区编号
//dst: 读出的数据存到地址为dst的地方去
//nescs: 待操作的扇区数量
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);
//diskno: 待操作的磁盘编号
//secno: 第一个待操作的扇区编号
//src: 把地址src处存的内容写到磁盘指定区域去
//nescs: 待操作的扇区数量
```

注意所有文件系统中调用的系统调用函数，无论是内核态还是用户态，都只能调用 `syscall_*` 类型，因为他们才是封装好的接口，`sys_*` 类型是没有对应调用接口的

硬件外设与读写驱动

fsformat.c / create_file

文件控制块 - 256B

磁盘块 - 4KB

扇区 - 512B

函数功能：该函数创建一个文件的镜像，也就是准备好文件内容，但是还不会把他真正加载到磁盘块上，也必然不会加载到内存中

```
/* Overview
  函数功能 : 在目录 firf 之下，找到或创建一个空闲的 struct File 结构，用来表示一个新的文件
  注意 : 该函数[永远不会失败]，因为如果不存在空闲的 File 项，会调用 make_link_block 给目录扩展一个新的块
*/
struct File *create_file(struct File *dirf) {
// 首先遍历 dirf 目录下所有已有的磁盘块
        int nblk = dirf->f_size / BLOCK_SIZE;
        for (int i = 0; i < nblk; ++i) {
                int bno;
          // 直接块
                if (i < NDIRECT) {
                        bno = dirf->f_direct[i];
                } else {
                  // 间接块
                        bno = ((int *)(disk[dirf->f_indirect].data))[i];
                }
// 把块号为 bno 的磁盘块从 disk 数组中读出来，并且解释成一个 File 类型数组(因为目录文件由目录块组成，目录块由目录项组成，目录项就是 File 一个个结构体)
                struct File *blk = (struct File *)(disk[bno].data);

// 遍历当前目录块中的每一个目录项
                for (struct File *f = blk; f < blk + FILE2BLK; ++f) {
                        // 如果找到空闲的目录项，也就是没分配过名字，就直接返回
```

```
                        if (f->f_name[0] == '\0') {
                                return f;
                        }
                }
        }

// 如果在已有目录下找不到空闲的目录项，就为他扩展一个新块，返回新分配的目录块中第一个目录项
        int bno = make_link_block(dirf, nblk);
        return (struct File *)(disk[bno].data);
        return NULL;
}

```

本次实验中，我们对所有硬件外设的访问都是通过**向内存中某个空间写入**来实现的，具体来讲，我们要访问的外设就是**控制台 console** 和**硬盘 IDE disk**，首先我们要实现提供这些外设使用的读写驱动（接口），才能在后面的文件系统中访问这些外设

```
* ----------------------------------*
|   设备名    | 起始物理地址 |  长度   |
* -----------+------------+--------*
|  console   | 0x180003f8 |  0x20   |
|  IDE disk  | 0x180001f0 |  0x8    |
* ----------------------------------*
```

硬件外设的访问

在 MIPS 的内核地址空间中（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，且不使用高速缓存，这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

物理地址 -> kseg0 段内核虚拟地址：加上 0x80000000 偏移量

物理地址 -> kseg1 段内核虚拟地址：加上 0xA0000000 偏移量

内核态实现

在内核态下，可以直接对内核段虚拟地址进行访问，所以我们可以直接用 kseg1 段的地址作为偏移量，访问该段内核虚拟地址

以下是函数 `printcharc` 两种等价的函数实现形式，由此可知有以下地址等式：

`MALTA_SERIAL_DATA（数据寄存器地址） = DEV_CONS_ADDRESS（设备基地址） + DEV_CONS_PUTGETCHAR（数据寄存器相对于设备的偏移量）`

```
void printcharc(char ch) {
    *((volatile char *)(KSEG1 + DEV_CONS_ADDRESS + DEV_CONS_PUTGETCHAR)) = ch;
}
void printcharc(char ch) {
  *((*volatile char *)(KSEG1 + MALTA_SERIAL_DATA)) = ch;
}
```

用户态实现

由于我们采用微内核的思路进行设计，所以文件系统需要工作于用户态中，那么就必须要实现用户态中的读写操作，MOS 采用**系统调用**的方式，先**陷入内核态**，再直接利用上一种方法读写，最后完成信息等的传递

这里需要我们实现和硬件设备读和写有关的两个系统调用函数，sys_write_dev 和 sys_read_dev 函数:

```
/* Overview:
  函数功能 : 把从虚拟地址 va 开始，长度为 len 的数据写入物理地址 pa 所对应的硬件设备
  前置条件 : 需要检查 va pa len 的合法性
    1. va 是源数据的起始地址，len 是数据长度(以字节为单位)
    2. pa 是设备的物理地址
    3. len 必须是 1 2 4 的倍数，否则视为不合法
  后置条件 :
    1. 把 [va, va + len) 范围内的数据复制到设备的物理地址 pa
    2. 成功时返回 0，若地址不合法返回错误码
  以下是合法设备以及物理地址的范围(其实就两个设备，console 和 IDE disk) :
       * ---------------------------------*
       |    设备名    | 起始物理地址 |  长度   |
       * -----------+------------+--------*
       |  console   | 0x180003f8 | 0x20    |
       |  IDE disk  | 0x180001f0 | 0x8     |
       * ---------------------------------*
 */
int valid_addr_space_num = 2;// 合法设备地址空间的数量 - 只有 2 个
unsigned int valid_addr_start[2] = {0x180003f8, 0x180001f0};// 合法设备的起始地址
unsigned int valid_addr_end[2] = {0x180003f8 + 0x20, 0x180001f8};// 合法设备的结束地址

/* Overview:
  函数功能 : 内联函数，检查设备物理地址是否合法，对应先前检验虚拟地址合法性的 is_illegal_va 函数
  入参含义 : 判断设备物理地址 pa 和地址长度 len 是否构成非法地址范围
*/
static inline int is_illegal_dev_range(u_long pa, u_long len) {
/* 检查对齐合法性 :
  1. 如果 pa 不是 4 字节对齐，却要读写 4 字节，非法
  2. 如果 pa 不是 2 字节对齐，却要读写 2 字节，非法
*/
        if ((pa % 4 != 0 && len != 1 && len != 2) || (pa % 2 != 0 && len != 1)) {
                return 1;
        }
        int i;
        u_int target_start = pa;// 起始物理地址
        u_int target_end = pa + len;// 结束物理地址
// 检查 [pa, pa + len) 是否完全落在某个合法设备的地址范围内，如果可以找到这样的设备，就立即返回，反之则说明不存在
        for (i = 0; i < valid_addr_space_num; i++) {
                if (target_start >= valid_addr_start[i] && target_end <= valid_addr_end[i]) {
                        return 0;
                }
        }
        return 1;
```

```c
}

// 主函数实现，入参含义 : va 虚拟地址，len 长度，pa 物理地址
int sys_write_dev(u_int va, u_int pa, u_int len) {
/*
  检查地址对齐的合法性 :
    1. 虚拟地址范围是否合法(is_illegal_va_range)
    2. 物理地址范围是否合法，同时也检验是否长度对齐(is_illegal_dev_range)
    3. 虚拟地址是否按长度对齐
*/
        if (is_illegal_va_range(va, len) || is_illegal_dev_range(pa, len) || va % len != 0) {
                return -E_INVAL;
        }
/*
  根据读取数据的长度，调用不同的函数，把数据写入对应的物理地址，其中 iowrite32，iowrite16，iowrite8 用于向设备的寄存器写入数据
  以 iowrite16(*(uint16_t *)va, pa) 为例分析 :
    1. (uint_16 *va) : 把 va 强制转化成 u_int16 * 类型的指针，这里考虑指针步长，确保解引用能取得 uint_16 大小的数据
    2. *(uint_16 *va) : 表示对这个指针解引用，取得当前地址处的 16 位值
    3. 最后把这 16 位值写入 I/O 设备区域的 pa 这个物理地址的空间
*/
        if (len == 4) {
                iowrite32(*(uint32_t *)va, pa);
        } else if (len == 2) {
                iowrite16(*(uint16_t *)va, pa);
        } else if (len == 1) {
                iowrite8(*(uint8_t *)va, pa);
        } else {
                return -E_INVAL;
        }
        return 0;
}

/* Overview:
  函数功能 : 从设备的物理地址处读取数据
  前置 / 后置条件 : 同 sys_write_dev 即可
 */
int sys_read_dev(u_int va, u_int pa, u_int len) {
// 完全仿照 sys_write_dev 实现即可，只是把所有 iowrite 都换成 ioread
        if (is_illegal_va_range(va, len) || is_illegal_dev_range(pa, len) || va % len != 0) {
                return -E_INVAL;
        }
        if (len == 4) {
                *(uint32_t *)va = ioread32(pa);
        } else if (len == 2) {
                *(uint16_t *)va = ioread16(pa);
        } else if (len == 1) {
                *(uint8_t *)va = ioread8(pa);
        } else {
                return -E_INVAL;
        }
        return 0;
}
```

在实现这两个内核系统调用函数的同时，也需要完成用户态下对应系统调用的接口，也就是在 user / lib / syscall_lib.c 中完成对应用户态下的 syscall_write_dev 和 syscall_read_dev 函数：

```
int syscall_write_dev(void *va, u_int dev, u_int size) {
        return msyscall(SYS_write_dev, va, dev, size);
}

int syscall_read_dev(void *va, u_int dev, u_int size) {
        return msyscall(SYS_read_dev, va, dev, size);
}
```

IDE 磁盘与读写

在 MOS 系统中，GXemul 为我们提供了模拟的 IDE 磁盘，每次读写的单位是一个扇区（512B），和其他的外设一样，在指定位置读写可以实现和磁盘的交互，地址表如下：

偏移	寄存器功能	数据位宽
0x0	读/写：向磁盘中读/写数据，从 0 字节开始逐个读出/写入	4 字节
0x1	读：设备错误信息；写：设置 IDE 命令的特定参数（实验中不涉及）	1 字节
0x2	写：设置一次需要操作的扇区数量	1 字节
0x3	写：设置目标扇区号的 [7:0] 位（LBAL）	1 字节
0x4	写：设置目标扇区号的 [15:8] 位（LBAM）	1 字节
0x5	写：设置目标扇区号的 [23:16] 位（LBAH）	1 字节
0x6	写：设置目标扇区号的 [27:24] 位，配置扇区寻址模式（CHS/LBA），设置要操作的磁盘编号	1 字节
0x7	读：获取设备状态；写：配置设备工作状态	1 字节

在进行磁盘 I/O 操作时，CPU 通过访问特定的 I/O 端口偏移（0x0 到 0x7），对硬盘进行读写操作或者状态控制，其中**偏移**这个字段描述的是相对于 IDE 控制器的基地址的偏移地址，也就是由 Base + Offset 所得到的地址就是具体的端口地址

每个寄存器的作用详解

偏移	名称	作用说明
`0x0`	数据寄存器 (Data)	用于读/写磁盘数据。一次读/写 4 字节，数据传输就是通过它进行的。
`0x1`	错误寄存器 (读) / 特定参数 (写)	读时获取错误代码；写入特定命令参数（实验中不涉及）。
`0x2`	扇区数	设置每次操作的扇区数量（最多 256）。
`0x3`	LBAL（LBA 低 8 位）	目标扇区的 LBA 地址低 8 位。
`0x4`	LBAM（中间 8 位）	LBA 地址的中间 8 位。
`0x5`	LBAH（高 8 位）	LBA 地址高 8 位。
`0x6`	驱动器/磁头	设置第 4 字节的 LBA 地址高 4 位 + 选择主/从驱动器等信息。
`0x7`	状态寄存器 (读) / 命令寄存器 (写)	读时检查设备状态；写时发出操作命令（如读扇区、写扇区）。

读操作后需要从缓冲区取出数据，写操作前需要实现写入缓冲区内

内核态实现

课程组在指导书中给出了内核态的访问实现，但实际上我们的 MOS 中并不需要这一函数去实现驱动，因为我们采用微内核架构，将访问磁盘的操作交给**用户态**完成，这里的代码只起到示例作用

```
// 定义 - C
extern int read_sector(int diskno, int offset);
// 实现 - MIPS
LEAF(read_sector)
    sw a0, 0xB3000010  # choose the IDE id, must 0 in our MOS
    sw a1, 0xB3000000  # offset in the disk
    li t0, 0
    sw t0, 0xB3000020  # launch a 'read' action
    lw v0, 0xB3000030  # get the result of the action
    nop
    jr ra
    nop
END(read_sector)
```

用户态实现

在 MOS 中，实际完成对磁盘读写操作的是 `fs/ide.c` 中的两个函数 ide_read 和 ide_write，他们可在用户态使用，充当磁盘驱动，在这里我们就相当于复刻了前面的内核态驱动，使用 C 语言替代 MIPS，并使用系统调用来完成写入地址的操作，下面是这两个函数的具体实现：

```
/* Overview:
  函数功能 : 等待 IDE 设备完成前一个请求，并且准备好接受下一个请求
 */
static uint8_t wait_ide_ready() {
        uint8_t flag;
// 持续轮旋 IDE 设备的状态寄存器，直到设备不再忙碌(意味着春备好新的读写请求)为止，用于同步 CPU 和设备的状态
```

```
        while (1) {
                panic_on(syscall_read_dev(&flag, MALTA_IDE_STATUS, 1));
                if ((flag & MALTA_IDE_BUSY) == 0) {
                        break;
                }
                syscall_yield();
        }
        return flag;
}

/* Overview:
  函数功能 : 从指定的 IDE 磁盘中读取多个扇区的数据到 dst 缓冲区中
  入参含义 :
    1. dikno : 磁盘编号(0 或 1)
    2. secno : 起始扇区号
    3. dst : 从 IDE 磁盘读取数据的目标地址
    4. nsecs : 要读取的扇区数量
  后置条件 : 所有错误都由 panic 来触发
 */
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs) {
        uint8_t temp;// 临时变量，用于传递单字节数据到寄存器
        u_int offset = 0;// 读取过程中，目标地址的偏移量(初始为 0)
        u_int max = nsecs + secno;// 计算最后一个扇区号+1(用于循环终止条件)
        panic_on(diskno >= 2);

        // 循环读取每个扇区，从 secno 到 max-1
        while (secno < max) {
       // 调用 wait_ide_ready() 函数等待 IDE 控制器就绪，该函数内部会检查状态寄存器直到设备就绪
                temp = wait_ide_ready();
       // 设置扇区数量，每次操作读取一个扇区，使用 syscall_write_dev 系统调用写入 NSECT 寄存器
                temp = 1;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_NSECT, 1));

        // 设置扇区号低 8 位，将其提取出来并且写入 LBAL 寄存器
                temp = secno & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAL, 1));

        // 设置扇区号中 8 位，将其提取出来并且写入 LBAM 寄存器
                temp = (secno >> 8) & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAM, 1));

        // 设置扇区号高 8 位，将其提取出来并且写入 LBAH 寄存器
                temp = (secno >> 16) & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAH, 1));

        // 设置最高 4 位扇区号，LBA 模式和磁盘号，提取扇区号的高 4 位，设置 LBA 模式标志(MALTA_IDE_LBA)，设置磁盘号左移 4 位，组合后写入 DEVICE 寄存器
                temp = ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
                panic_on(syscall_write_dev(&temp, MALTA_IDE_DEVICE, 1));

           // 写入 PIO 模式下的读命令，向 TATUS 寄存器中写入 PIO 读取命令，这会启动磁盘的读取操作，注意 MATA_IDE_CMD_PIO_READ 是读取命令
                temp = MALTA_IDE_CMD_PIO_READ;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_STATUS, 1));

```

```
        // 再次等待 IDE 就绪，也就是磁盘操作完成
                temp = wait_ide_ready();

         // 读取数据，循环读取 512 字节(SECT_SIZE)的数据，每次读取 4 字节，共读取 128 次，每次读取的数据被复制到 dst + offset 的位置(从设备相内存传输数据)
                for (int i = 0; i < SECT_SIZE / 4; i++) {
                        panic_on(syscall_read_dev(dst + offset + i * 4, MALTA_IDE_DATA, 4));
                }

                // 检查 IDE 的状态，通过读取状态寄存器确认操作成功
                panic_on(syscall_read_dev(&temp, MALTA_IDE_STATUS, 1));
               // 更新偏移和扇区号
                offset += SECT_SIZE;
                secno += 1;
        }
}

// 第三个参数从 dst 变成 src，表示从写入目标地址(外存)变成了数据来源地址(内存)
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs) {
        uint8_t temp;// 临时变量，用于传递单字节数据到寄存器
        u_int offset = 0;// 读取过程中，目标地址的偏移量(初始为 0)
        u_int max = nsecs + secno;// 计算最后一个扇区号+1(用于循环终止条件)
        panic_on(diskno >= 2);

// 循环读取每个扇区，从 secno 到 max-1
        while (secno < max) {
        // 调用 wait_ide_ready() 函数等待 IDE 控制器就绪，该函数内部会检查状态寄存器直到设备就绪
                temp = wait_ide_ready();
        // 设置扇区数量，每次操作读取一个扇区，使用 syscall_write_dev 系统调用写入 NSECT 寄存器
                temp = 1;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_NSECT, 1));

        // 设置扇区号低 8 位，将其提取出来并且写入 LBAL 寄存器
                temp = secno & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAL, 1));

        // 设置扇区号中 8 位，将其提取出来并且写入 LBAM 寄存器
                temp = (secno >> 8) & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAM, 1));

        // 设置扇区号高 8 位，将其提取出来并且写入 LBAH 寄存器
                temp = (secno >> 16) & 0xff;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAH, 1));

        // 设置最高 4 位扇区号，LBA 模式和磁盘号，提取扇区号的高 4 位，设置 LBA 模式标志(MALTA_IDE_LBA)，设置磁盘号左移 4 位，组合后写入 DEVICE 寄存器
                temp = ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
                panic_on(syscall_write_dev(&temp, MALTA_IDE_DEVICE, 1));

        // 写入 PIO 模式下的读命令，向 TATUS 寄存器中写入 PIO 读取命令，这会启动磁盘的读取操作，注意 MALTA_IDE_CMD_PIO_WRITE 是写入命令
                temp = MALTA_IDE_CMD_PIO_WRITE;
                panic_on(syscall_write_dev(&temp, MALTA_IDE_STATUS, 1));

```

```
111         // 再次等待 IDE 就绪，也就是磁盘操作完成
112                 temp = wait_ide_ready();
113 
114          // 写入数据 : 每次从内存中读出 4 个字节的数据，写入 IDE 控制器的数据寄存器的物理地址，硬件会自动把数据存入磁盘缓冲区(从内存向设备传入数据)
115                 for (int i = 0; i < SECT_SIZE / 4; i++) {
116                         panic_on(syscall_write_dev(src + offset + i * 4, MALTA_IDE_DATA, 4));
117                 }
118 
119                 // 检查 IDE 的状态，通过读取状态寄存器确认操作成功
120                 panic_on(syscall_read_dev(&temp, MALTA_IDE_STATUS, 1));
121                // 更新偏移和扇区号
122                 offset += SECT_SIZE;
123                 secno += 1;
124         }
125 }
```

需要注意的是，在函数中设置读写的偏移量时，这里的偏移量不是 off 提供的偏移，而是读写地址相对于磁盘起始的偏移，也就是 `secno * BY2SECT + off`

完成这两个函数后，我们就可以通过在用户态调用 `ide_read/write` 来对磁盘进行操作了，这里的读写操作并不限制我们读写的大小（扇区数），但一定是**整数个扇区（section）**，通过对磁盘的自由读写，我们就能在此基础上建立合适的文件系统了

磁盘结构

需要区分磁盘的两种常见空间单位：

1. 扇区（section）：对磁盘空间的物理划分（实际划分），大小通常为 512B
2. 磁盘块（block）：为了便于磁盘管理而设置的虚拟划分单位，和虚拟存储中的页相对应，大小为 4KB（一个页的大小）

Block 的定义如下（disk 数组为 Block 类型）：

```
struct Block {
  uint8_t data[BLOCK_SIZE];
  uint32_t type;
} disk[NBLOCK];
```

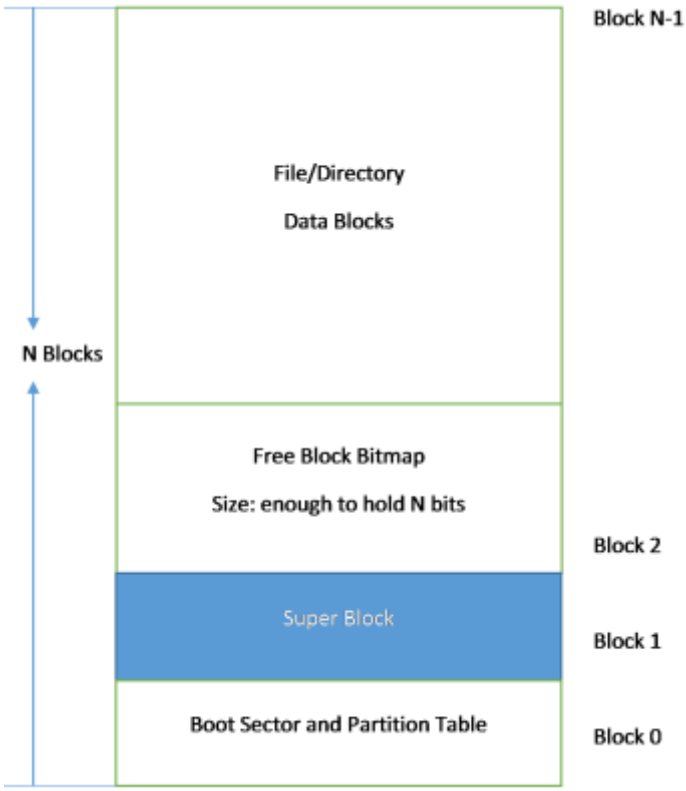
通常情况下我们以磁盘块作为逻辑上管理磁盘的单位，每个 Block 的大小为 4100 字节

其次是两个特殊的磁盘块，分别代表磁盘上的第 0 块和第 1 块：引导扇区 + 分区表、超级块（Super Block）

- 引导扇区+分区表：0 号磁盘块，我们只需要在概念上预留出这部分空间，功能上并不需要真正实现
- 超级块（Super）：1 号磁盘块，包含磁盘的一些判断信息，同时充当**当前磁盘的根目录**

磁盘块大小由操作系统决定（这里一个磁盘块的大小为 4096B，和一个**页面**大小相同），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

磁盘空间布局示意图：



超级块 Super 的结构如下：

```
struct Super {
  u_int s_magic;  // 魔数，一个常量，用来标识该文件系统
  u_int s_nblocks;  // 记录文件系统中有多少个磁盘块，这里为 1024
  struct File s_root; // 根目录，file_type 为 FTYPE_DIR，file_name 为 /
}
```

在 MOS 中，我们通过设置结构体 Block 类型的数组 disk 来对磁盘进行表示，每个 Block 就代表一个磁盘块，data 是具体的空间，type 是磁盘块用途和状态的标记

磁盘状态记录

在实验中的 IDE 磁盘中，我们占用了一些磁盘块用来充当标记是否已分配磁盘块的**位图数组**，通过读取数组的状态来得知磁盘块的分配情况，这点与内存管理中我们使用的链表不同，通常情况下这部分磁盘块紧跟磁盘的**超级块**

我们采用**位图**（bitmap）管理空闲的磁盘资源，即用一个二进制位 bit 表示磁盘中的一个磁盘块的使用情况（1 表示空闲，0 表示占用），具体来讲，bitmap 是存在于外存中的，且位置就在超级块 super 之后

在 tools / fsformat.c / disk_init 中，记录了磁盘块初始化的过程：

```
// 函数功能 : 初始化整个磁盘(也就是 disk[] 数组)，包括 boot sector，bitmap，superblock
void init_disk() {
        int i, diff;

// 标记引导扇区，设置 0 号块为引导扇区 BLOCK_BOOT
        disk[0].type = BLOCK_BOOT;
```

```
// 计算 bitmap 所需块数，NBLOCK 是磁盘总块数，BLOCK_SIZE 是一个磁盘块的大小(单位是字节)，BLOCK_SIZE_BIT 是一个磁盘块所包含的位数(1B == 8bit)，这里我们用一个 bit 位表示一个磁盘块，而一个磁盘块中有 BLOCK_SIZE_BIT 个位，总共要表示 NBLOCK 个磁盘块，所以就需要 NBOCK / BLOCK_SIZE_BIT 这么多个磁盘块，最后记得[向上取整]
// 这里之所以要按磁盘块算，是因为 bitmap 也是存在外存中，我们用 2 到 nbitblock 这个范围内的磁盘块存储 bitmap(0 是引导块，1 是超级块)，这样 nextbno 就是第一个用来存放普通文件或目录的块号
        nbitblock = (NBLOCK + BLOCK_SIZE_BIT - 1) / BLOCK_SIZE_BIT;
        nextbno = 2 + nbitblock;

// 设置所有存放 bitmap 的块的类型为 BLOCK_BMAP
        for (i = 0; i < nbitblock; ++i) {
                disk[2 + i].type = BLOCK_BMAP;
        }
// 初始化所有 bitmap 为 1
        for (i = 0; i < nbitblock; ++i) {
                memset(disk[2 + i].data, 0xff, BLOCK_SIZE);
        }
// 因为先前是向上取整，有可能最后一块 bitmap 中有多余的位，我们需要把这些多余的位清零
        if (NBLOCK != nbitblock * BLOCK_SIZE_BIT) {
                diff = NBLOCK % BLOCK_SIZE_BIT / 8;
                memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BLOCK_SIZE - diff);
        }

// 初始化超级块，把 1 号块标记为超级块(Super Block)，设置关键字段(魔数，磁盘块数，根目录类型，根目录名称)
        disk[1].type = BLOCK_SUPER;
        super.s_magic = FS_MAGIC;
        super.s_nblocks = NBLOCK;
        super.s_root.f_type = FTYPE_DIR;
        strcpy(super.s_root.f_name, "/");
}
```

磁盘文件的创建

我们使用 `tools/fsformat.c` 的 `main` 函数来在 Linux 中创建一个可供我们 MOS 使用的磁盘镜像文件，`Usage: fsformat <img-file> [files or directories]...` 意味着函数的 `argv[1]` 代表着生成镜像的路径，后面的每个参数都代表准备写入的文件/目录路径

我们来分析一下 `fsformat.c` 是怎样创建一个磁盘镜像的

```
int main(int argc, char **argv) {
// 静态断言，确保 File 结构体大小和文件系统规定的大小一致
        static_assert(sizeof(struct File) == FILE_STRUCT_SIZE);
// 初始化磁盘数据结构
        init_disk();
// 参数不足，直接退出
        if (argc < 3) {
                fprintf(stderr, "Usage: fsformat <img-file> [files or directories]...\n");
                exit(1);
        }
// 遍历命令行传入的每个文件和目录
        for (int i = 2; i < argc; i++) {
                char *name = argv[i];
                struct stat stat_buf;
```

```
                int r = stat(name, &stat_buf);
                assert(r == 0);
                // 如果是目录，递归调用 write_directory 写入磁盘
                if (S_ISDIR(stat_buf.st_mode)) {
                        printf("writing directory '%s' recursively into
disk\n", name);
                        write_directory(&super.s_root, name);
                // 如果是常规文件，直接调用 write_file 写入磁盘
                } else if (S_ISREG(stat_buf.st_mode)) {
                        printf("writing regular file '%s' into disk\n",
name);
                        write_file(&super.s_root, name);
                // 非法情况直接报错并退出
                } else {
                        fprintf(stderr, "'%s' has illegal file mode %o\n",
name, stat_buf.st_mode);
                        exit(2);
                }
        }
// 把文件系统的位图写入磁盘，最后把文件系统的结构写入镜像文件
        flush_bitmap();
        finish_fs(argv[1]);

        return 0;
}
```

这里我们用到了 main 函数的入参，各个入参的含义如下：

索引	值（argv[i]）	含义
`0`	`"fsformat"`	程序本身名称（通常无实际用处）
`1`	`"myimg.img"`	要创建的文件系统镜像文件名（输出文件）
`2...5`	`"dir1" / "file.txt"`	要写入的目录或文件名

其中 S_ISDIR 和 S_ISREG 是两个宏函数：

```
struct stat stat_buf; // 入参类型 : 文件属性信息结构体
S_ISDIR(stat_buf.st_mode); // 判断文件是否为目录类型
S_ISREG(stat_buf.st_mode); // 判断文件是否为普通文件类型
```

写入前初始化

在 init_disk 函数中，我们对位图和特殊的磁盘块进行了初始化：

1. 初始化引导扇区块和超级块
2. 把所有位图需要的磁盘标记为 BMAP 类
3. 对位图所对应的磁盘块的 data 区域进行初始化，设置为可用
4. 修订位图数组，删除因为向上取整而被多初始化的那些磁盘块

```
void init_disk() {
        int i, diff;

        disk[0].type = BLOCK_BOOT;

```

```
        nbitblock = (NBLOCK + BLOCK_SIZE_BIT - 1) / BLOCK_SIZE_BIT;
        nextbno = 2 + nbitblock;

        for (i = 0; i < nbitblock; ++i) {
                disk[2 + i].type = BLOCK_BMAP;
        }

        for (i = 0; i < nbitblock; ++i) {
                memset(disk[2 + i].data, 0xff, BLOCK_SIZE);
        }

        if (NBLOCK != nbitblock * BLOCK_SIZE_BIT) {
                diff = NBLOCK % BLOCK_SIZE_BIT / 8;
                memset(disk[2 + (nbitblock - 1)].data + diff, 0x00,
BLOCK_SIZE - diff);
        }

        disk[1].type = BLOCK_SUPER;
        super.s_magic = FS_MAGIC;
        super.s_nblocks = NBLOCK;
        super.s_root.f_type = FTYPE_DIR;
        strcpy(super.s_root.f_name, "/");
}
```

这里我们不需要真正实现引导扇区的功能，因此只是做了状态的标记，而超级块是我们要具体实现的，需要进行有关字段的初始化

文件信息获取

struct stat 这个结构体是用来描述一个 Linux 系统文件系统中的文件属性的结构，可以通过 stat 函数获取文件的所有相关信息，一般情况下，我们只关心文件的**大小**，**创建时间**，**访问时间**，**修改时间**

```
// 函数功能 : 通过[文件路径]来获取文件状态信息，path 为要查询的文件路径，buf 为用于保存文件信息的结构体指针，获取成功则返回 0
int stat(const char *path, struct stat *buf);
// 函数功能 : 通过[文件描述符]来获取文件状态信息，sdnum 为文件描述符，stat 为用于保存文件信息的结构体指针，获取成功则返回 0
int fstat(int fdnum, struct Stat *stat);

struct stat {
        mode_t     st_mode;
        ino_t      st_ino;
        dev_t      st_dev;
        dev_t      st_rdev;
        nlink_t    st_nlink;
        uid_t      st_uid;
        gid_t      st_gid;
        off_t      st_size;
        time_t     st_atime;
        time_t     st_mtime;
        time_t     st_ctime;
        blksize_t st_blksize;
        blkcnt_t   st_blocks;
      };
```

成员	类型	含义	示例 / 补充
`mode_t st_mode`	文件模式	包含文件类型和访问权限（rwx）	可使用宏如 `S_ISDIR(st_mode)` 判断是否为目录；或掩码 `st_mode & 0777` 查看权限
`ino_t st_ino`	inode 号	文件的索引节点编号，唯一标识一个文件	在 ext4 文件系统中，每个文件都有唯一 inode
`dev_t st_dev`	所在设备 ID	表示文件所在的设备编号	如 `/dev/sda1` 的设备号
`dev_t st_rdev`	设备类型	仅对设备文件有效（如 `/dev/tty`）	通常不用关心，除非操作字符/块设备文件
`nlink_t st_nlink`	硬链接数	指向该 inode 的目录项数量	删除一个硬链接不等于删除文件内容，除非此值变为 0
`uid_t st_uid`	所有者用户 ID	所属用户 ID	用于判断访问权限等
`gid_t st_gid`	所有者组 ID	所属用户组 ID	
`off_t st_size`	文件大小（字节）	文件的实际内容大小	常用于读取文件
`time_t st_atime`	最后访问时间	上次读取文件内容的时间（read）	精度可能受文件系统限制
`time_t st_mtime`	最后修改时间	上次修改文件内容的时间（write）	文件内容改变时更新
`time_t st_ctime`	状态改变时间	文件的权限、所有者等元数据发生变化的时间	注意不是“创建时间”
`blksize_t st_blksize`	块大小	文件系统建议用于 I/O 的块大小	对于优化读写性能有帮助
`blkcnt_t st_blocks`	占用磁盘块数	文件实际占用了多少磁盘块（单位：512 字节）	与 `st_size` 不一定对应

文件写入

根据 S_ISDIR 和 S_ISREG 这两个宏函数的判断结果，我们分别为**目录文件**和**常规文件**设置了两种写入函数，借助这两个函数，我们可以把待写入的文件在 Linux 环境中转写到磁盘镜像中，他们就是 write_directory 函数和 write_file 函数：

```
// Overview:
// 函数功能 : 把一个目录写入到磁盘中的指定目录下，入参含义 : 父目录 dirf，文件路径 path
```

```c
void write_directory(struct File *dirf, char *path) {
// 使用 POSIX 标准函数 opendir 打开 path 对应的目录，成功时返回一个 DIR * 类型的指针，可以用来读取该目录下的所有文件项
  DIR *dir = opendir(path);
  if (dir == NULL) {
    perror("opendir");
    return;
  }
// 在磁盘结构中，通过父目录 dirf 创建一个新的目录文件项，通过调用 create_file 函数为当前目录创建一个子目录项，并且分配空间
  struct File *pdir = create_file(dirf);
// 设置新创建目录文件的名称，basename 返回路径的最后部分，即目录的名字，MAXNAMELEN - 1 为拷贝长度，防止越界
  strncpy(pdir->f_name, basename(path), MAXNAMELEN - 1);
  // 如果长度越界。也就是拷贝名称没能以 \0 结尾，直接报错
  if (pdir->f_name[MAXNAMELEN - 1] != 0) {
    fprintf(stderr, "file name is too long: %s\n", path);
    exit(1);
  }
  // 设置新文件项的类型为目录
  pdir->f_type = FTYPE_DIR;
  // 循环读取目录中的每一项，也就是每一个子文件或子目录，其中 readdir 函数每次返回一个 struct dirent * 指针，直到读完为止
  for (struct dirent *e; (e = readdir(dir)) != NULL;) {
    // 跳过当前目录 . 和上级目录 ..
    if (strcmp(e->d_name, ".") != 0 && strcmp(e->d_name, "..") != 0) {
      // 构建当前目录项的完整路径，由 path 和当前目录名称组成，+2 是为了中间拼接的 / 和最后的 \0
      char *buf = malloc(strlen(path) + strlen(e->d_name) + 2);
      sprintf(buf, "%s/%s", path, e->d_name);
      // 如果当前项还是子目录，就递归调用 write_directory 继续写入，如果是常规文件，就调用 write_file 写入文件内容
      if (e->d_type == DT_DIR) {
        write_directory(pdir, buf);
      } else {
        write_file(pdir, buf);
      }
      // 释放记录文件名称的 buf 资源
      free(buf);
    }
  }
  // 关闭当前目录，释放资源
  closedir(dir);
}

// Overvew
// 把文件写入到磁盘的指定目录下
void write_file(struct File *dirf, const char *path) {
// iblk 表示文件的第几个逻辑块，r 表示每次实际读取的字节数，n 表示每块的大小，也就是 distk[0].data 的容量
  int iblk = 0, r = 0, n = sizeof(disk[0].data);
// 在父目录 dirf 下创建一个文件项 target，调用 create_file 分配一个 struct File 结构体，并且与父目录 dirf 建立关系
  struct File *target = create_file(dirf);

// 建立失败返回
  if (target == NULL) {
    return;
```

```
  }

// 调用 open 函数以只读模式打开文件，若成功则返回文件描述符 fd
  int fd = open(path, O_RDONLY);

// 从完整路径 path 中提取出文件名，strrchr 可以返回 path 中最后一个 / 出现的位置，如果找到就把指针后移一位，跳过 /，反之则说明 path 本身就是文件名，直接使用即可
  const char *fname = strrchr(path, '/');
  if (fname) {
    fname++;
  } else {
    fname = path;
  }
// 把找到的文件名写入 target->f_name
  strcpy(target->f_name, fname);
// 移动文件指针到末尾，以获取文件大小，返回值以字节数为单位
  target->f_size = lseek(fd, 0, SEEK_END);
// 标记当前文件类型为普通文件
  target->f_type = FTYPE_REG;

// 把文件读指针重置到开头，为后续读入做准备
  lseek(fd, 0, SEEK_SET);
/* 文件读取的核心部分 :
  read(fd, disk[nextbno].data, n)
    1. read 函数可以从文件中读 n 个字节到模拟磁盘的某个块中          2. disk[nextbno].data 表示当前模拟磁盘的下一个空数据块
  save_block_link(target, iblk++, next_block(BLOCK_DATA))
    1. 把当前块与 target 文件逻辑编号为 iblk 的块进行绑定          2. next_block 分配下一个空闲数据块
    3. save-block_link 函数把文件逻辑块号映射到物理块号，实现数据定位
*/
  while ((r = read(fd, disk[nextbno].data, n)) > 0) {
    save_block_link(target, iblk++, next_block(BLOCK_DATA));
  }
  close(fd);
}
```

至此，我们完成了磁盘镜像的生成，此时磁盘可以通过 fsformat 中的程序顺利生成，接下来才是真正的文件管理的内容

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

在 MOS 系统中，文件管理系统，是以一个进程的形式存在的，它通过从磁盘中读写数据，并与其他用户进程交互来实现文件的管理，这些被使用到的文件就会先被**缓存**在文件服务进程的进程空间中

我们规定，文件管理进程使用大小为 `DISKMAX` 字节的空间作为磁盘块的缓存区，并且缓存区的存取**单位为磁盘块**。每个磁盘块都应该在内存中有**单独相对应的位置**进行缓存，这样就限制了我们内核支持的最大磁盘大小

```
#define DISKMAP 0x10000000
#define DISKMAX 0x40000000
```

为了在磁盘块和内存之间进行交换，我们需要准备一系列辅助和工作函数，而这系列的共 35 个函数都被放置在 `fs/fs.c` 文件中（fs 似乎是 file service 的简称）

struct Block

Block 结构体记录的是磁盘块，磁盘分块是一个虚拟概念，是操作系统与磁盘交互的最小单位，而扇区是一个物理概念，是磁盘进行读写的基本单位，与操作系统无关

Block 结构（tools / fsformat.c）

```
struct Block {
  uint8_t data[BLOCK_SIZE]; // 一个磁盘块大小为 4KB
  uint32_t type;// 磁盘块类型(目录块 or 数据块)
} disk[NBLOCK];
```

磁盘块操作函数 - fs.c

这些函数都是在 fs.c 中实现的函数

diskaddr

函数功能：和上文的块缓存相对应，返回某个特定块在文件管理系统进程内存中**应该被放置到**的虚拟地址，使用线性排列即可，用一个基地址 DIKMAP 加上偏移量，而偏移量正比于磁盘块的**块号**

```
// Overview:
/*
  函数功能 : 根据给定的磁盘块号 blockno 返回该磁盘块在缓存中的虚拟地址
*/
void *disk_addr(u_int blockno) {
// DISKMAP 表示磁盘缓存的起始虚拟地址，指向内核虚拟空间中用于映射磁盘块的起点；BLOCK_SIZE 表示一个磁盘块的大小；blockno 表示磁盘块编号，从 0 开始编号，第 n 块的起始地址 = 起始地址 + n × 每块大小
        return (void *)(DISKMAP + blockno * BLOCK_SIZE);
}
```

va_is_mapped

函数功能：检查文件管理进程中的某个虚拟地址是否被使用（也就是是否和**物理页面**发生了有效映射），通常不会被单独使用，而是结合 diskaddr 在 block_is_mapped 中使用

```
// Overview:
//  函数功能 : 检查虚拟地址 va 是否被映射到物理块(通过检查页表项的 PTE_V 有效位)
int va_is_mapped(void *va) {
// 分别检验页目录和页表中是否有有效的映射
        return (vpd[PDX(va)] & PTE_V) && (vpt[VPN(va)] & PTE_V);
}
```

block_is_mapped

函数功能：检查特定的磁盘块是否使用块缓存装入了内存，若已装入则返回其在进程空间中映射的虚拟地址，未装入则返回 Null

因块缓存中，磁盘块和虚拟地址是一一对应的，因此只要查找磁盘块对应的虚拟地址是否和某个物理地址有映射，就可以得知与之对应的磁盘块是否也映射了物理地址

```c
// Overview:
// 函数功能 : 检查指定的磁盘块是否已经映射到内存缓存中，如果已经映射，则返回对应的虚拟地址，否则返回 0
void *block_is_mapped(u_int blockno) {
        void *va = disk_addr(blockno);
        if (va_is_mapped(va)) {
                return va;
        }
        return NULL;
}
```

map_block

函数功能：为特定的磁盘块申请一个物理页进行映射（实际上是先通过线性映射，得到与磁盘块唯一对应的虚拟地址，在完成宏观虚拟地址和对应物理页的映射）

```c
// Overview:
//  给 blockno 对应的磁盘块分配一页内存(这里其实是磁盘块->虚拟地址->物理页的过程)
int map_block(u_int blockno) {
// 如果这个块已经映射到内存缓存中，则直接返回 0，表示无需重复分配
        if (block_is_mapped(blockno)) {
                return 0;
        }

// 如果没有映射，就通过系统调用申请一页内存，把该页映射到磁盘块对应的虚拟地址上
        return syscall_mem_alloc(0, disk_addr(blockno), PTE_D);
}
```

unmap_block

函数功能：和 map_block 互为对偶函数，有建立映射，就有取消映射的过程，也就是取消掉特定磁盘块和对应物理页的映射

```c
// Overview:
//  把 blockno 对应的磁盘块在内存中解除映射
void unmap_block(u_int blockno) {
// 使用 block_is_mapped 获取该磁盘块当前在内存中的物理页对应的虚拟地址(其实磁盘块和虚拟地址之间的映射反而更直接)
        void *va;
        va = block_is_mapped(blockno);

// 如果该磁盘块不是空闲的，是已经分配过的有效块，且在内存中被修改过(dirty)，则应该优先写回磁盘
        if (!block_is_free(blockno) && block_is_dirty(blockno)) {
                write_block(blockno);
        }

// 使用系统调用函数 syscall_mem_unmap 解除该磁盘块对应虚拟地址在内存中和物理块的映射
        panic_on(syscall_mem_unmap(0, va));

        user_assert(!block_is_mapped(blockno));
}
```

write_block

函数功能：把某个特定的磁盘块所对应的物理页中的内容**写回**到对应的磁盘块内（内存 -> 外存），注意这个 write 并不是写心新的内容，只是磁盘块在内存缓存中数据被**写回**磁盘块本身

```
// Overview:
//  函数功能 : 把该磁盘块在内存中的内容写回磁盘
void write_block(u_int blockno) {
        if (!block_is_mapped(blockno)) {
                user_panic("write unmapped block %08x", blockno);
        }

/* ide_write : 写磁盘操作(IDE 磁盘写函数)，参数含义
  1. 0 : 磁盘编号，假设只有一个 IDE 磁盘
  2. blockno * SECT2BLK : 对应的起始扇区号
  3. va : 写入内容所在的虚拟地址
  4. SECT2BLK : 每个块占几个扇区
*/
        void *va = disk_addr(blockno);
        ide_write(0, blockno * SECT2BLK, va, SECT2BLK);
}
```

read_block

函数功能：把某个特定磁盘块中的内容写入到其对应的物理页中（外存 -> 内存）

```
/* Overview
  函数功能 : 把特定磁盘块的内容写入内存的物理页中
  如果 bik != 0，设置 *bik 为该磁盘块在内存中的地址
  如果 isnew != 0，设置 *isnew 为 0
  若该块是此次从磁盘读进内存的，设置 *isnew 为 1
  (isnew 允许调用者如 file_get_block 在磁盘块被加载进内存的时候清除一些仅存在于内存的字段)
  入参含义 : 待加载到内存的磁盘块号 blockno，后两个相当于传值引用的记录变量
*/
int read_block(u_int blockno, void **blk, u_int *isnew) {
// 验证 blockno 的合法性，确保他在磁盘范围内，super 是超级块指针，super->s_nblocks 是磁盘总块数
        if (super && blockno >= super->s_nblocks) {
                user_panic("reading non-existent block %08x\n", blockno);
        }

// 验证该磁盘块是否被使用(不能是空闲的磁盘块)，bitmap 为位图，如果 bitmap 已经加载且该快是空闲的，则报错
        if (bitmap && block_is_free(blockno)) {
                user_panic("reading free block %08x\n", blockno);
        }

// 把块号转化为虚拟地址，disk_addr(blockno) 返回该块号在内存中的虚拟地址，这个地址是映射在用户地址空间中的一页
        void *va = disk_addr(blockno);

// 读取磁盘数据并且设置 *isnew
        if (block_is_mapped(blockno)) { // 已经在内存中
                if (isnew) {
                        *isnew = 0;
```

```
                }
        } else { // 本次才从磁盘读取
                if (isnew) {
                        *isnew = 1;
                }
                try(syscall_mem_alloc(0, va, PTE_D));
                ide_read(0, blockno * SECT2BLK, va, SECT2BLK);
        }

// 如果需要，把映射到的虚拟地址写入 blk
        if (blk) {
                *blk = va;
        }
}
```

block_is_free

函数功能：在 bitmap 中检查某个特定的磁盘块是否尚未被使用

```
// 通过位图 bitmap 检验编号为 blockno 的磁盘块是否是空闲的，1 表示是空闲，0 表示不是空闲(和 bitmap 的初始化对应起来)
int block_is_free(u_int blockno) {
// 非法条件 : super == 0 表示超级块指针为 null，说明文件系统没有初始化或者已经损坏；或者 blockno 已经超出了最大块的范围，都认为是非法访问，不认为是空闲块
        if (super == 0 || blockno >= super->s_nblocks) {
                return 0;
        }
        if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
                return 1;
        }
        return 0;
}
```

block_free

函数功能：把某个特定的磁盘块释放，对应修改其在 bitmap 中的标志位从 0 变为 1，再次回到空闲

```
// Overview:
//  函数功能 : 把位图中的某个块标记为空闲(标志位设成 0)
void free_block(u_int blockno) {
// 非法判断 : 如果 blockno 为 0 或者已经超过 super 中的总记录数，则直接返回
        if (blockno == 0 || blockno >= super->s_nblocks) {
                return;
        }

// 设置位图中 blockno 对应的标志位，表示该块空闲
        bitmap[blockno / 32] |= 1 << (blockno & 0x1f);
}
```

alloc_block_num

函数功能：申请一个空闲的磁盘块，在位图中寻找空闲块，若找到就将其分配，返回申请到的磁盘块号

```
// Overview:
// 在位图中查找一个空闲块并且将其分配，如果分配成功就返回这个块号，反之返回错误码
```

```
int alloc_block_num(void) {
        int blockno;
// 遍历所有的磁盘块，从 3 号块开始遍历，这是因为 0 号块一般是超级块，1 2 号块是 inode 区或目录块等保留区域
        for (blockno = 3; blockno < super->s_nblocks; blockno++) {
                if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
                // 如果该块是空闲，就标记为使用，并且返回对应块号，把修改后的 bitmap 写回磁盘中(因为 bitmap 本身就存在于外存中，我们在内存中修改对应要写回)
                        bitmap[blockno / 32] &= ~(1 << (blockno % 32));
                        write_block(blockno / BLOCK_SIZE_BIT + 2);
                        return blockno;
                }
        }
        // 没有空闲块，返回错误码
        return -E_NO_DISK;
}
```

alloc_block

函数功能：先申请一个空闲的磁盘块（调用上面的 alloc_block_num 函数），并且把申请到的空闲块在内存中建立**映射**，仍然返回申请到的空闲块号

```
// Overview:
//  分配一个块，首先在位图中查找一个空闲块，然后把他映射到内存中
int alloc_block(void) {
        int r, bno;
// 首先在位图中找到一个空闲块，如果成功就把分配到的块号存入 bno
        if ((r = alloc_block_num()) < 0) {
                return r;
        }
        bno = r;
// 把该块映射到内存
        if ((r = map_block(bno)) < 0) {
                free_block(bno);
                return r;
        }
// 返回分配成功的块号
        return bno;
}
```

read_super

函数功能：读取磁盘的超级块，获取其基础信息

```
// Overview:
// 函数功能 : 读取文件系统的超级块(super-block)，获取文件的基础信息
void read_super(void) {
        int r;
        void *blk;

// 读取超级块，从磁盘块号为 1 中读取数据，并把该块映射到内存，地址保存在 blk(一般 1 号是超级块，0 号是引导块，最后一个参数表示不需要设置为脏页，也就是没有写回标志)
        if ((r = read_block(1, &blk, 0)) < 0) {
                user_panic("cannot read superblock: %d", r);
        }

```

```
        super = blk;

// 验证魔数，检验超级块中的 s_magic 字段是否为 FS_MAGIC
        if (super->s_magic != FS_MAGIC) {
                user_panic("bad file system magic number %x %x", super->s_magic, FS_MAGIC);
        }

// 检验磁盘容量是否合理，验证超级块中的块总数是否超过磁盘最大容量限制
        if (super->s_nblocks > DISKMAX / BLOCK_SIZE) {
                user_panic("file system is too large");
        }

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

read_bitmap

函数功能：读取磁盘的位图(bitmap)，并获取相关基础信息

```
// Overview:
// 函数功能 : 读取并验证文件系统的位图，需要把所有位图块读入内存中，设置 bitmap 指向第一个位图块，对于每个位图块都检查他们是否标记为已使用状态
void read_bitmap(void) {
        u_int i;
        void *blk = NULL;

/* 计算位图块数量，并且将其读入内存
  nblocks 表示文件系统中总的数据块数量，BLOCK_SIZE_BIT 表示一个位图块可以记录多少个数据块，因此 nbitmap 为二者相除上取整，得到最大位图块数
  位图块从第二块开始，使用 read_block 函数把每一个位图块从外存读入到内存
  最后把全局变量 bitmap 设置为磁盘第二块的虚拟地址，也就是位图的起始虚拟地址(bitmap 指针是一个虚存概念)
*/
        u_int nbitmap = super->s_nblocks / BLOCK_SIZE_BIT + 1;
        for (i = 0; i < nbitmap; i++) {
                read_block(i + 2, blk, 0);
        }

        bitmap = disk_addr(2);

// 确保预留块已经被标记为已使用，也就是 0 和 1 号块
        user_assert(!block_is_free(0));
        user_assert(!block_is_free(1));

// 确保所有位图块本身也都标记为已使用(这里判断的是用来存储位图本身的那些块，要被标记为已使用，而不是位图中表示的那些块)
        for (i = 0; i < nbitmap; i++) {
                user_assert(!block_is_free(i + 2));
        }

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

在这两个函数中，我们完成了文件管理进程的**基础数据准备**：读取了磁盘的超级块（获得基础信息），同时得到了磁盘的位图 `bitmap`（占用情况）

va_is_dirty

函数功能：检查某个虚拟地址对应的物理页是否被修改过，通过页表对应的标志位检查即可

```
// Overview:
//  检查虚拟地址 va 是否为脏页(是否被修改)，通过检查页表项中的 PTE_DIRTY 位
int va_is_dirty(void *va) {
        return vpt[VPN(va)] & PTE_DIRTY;
}
```

block_is_dirty

函数功能：检查某个磁盘块对应的物理页是否被修改过，还是通过先找到这个磁盘块对应的虚拟地址 va，再调用 va_is_dirty 对虚拟地址进行检查即可

```
// Overview:
//  检查 blockno 编号对应的磁盘块是否为脏块，通过检查它对应的虚拟地址是否为脏页实现
int block_is_dirty(u_int blockno) {
        void *va = disk_addr(blockno);
// 磁盘块映射了虚拟地址 va，并且 va 有 PTE_DIRTY 标志
        return va_is_mapped(va) && va_is_dirty(va);
}
```

dirty_block

函数功能：对某个磁盘块所对应的物理页进行写入修改，也就是设置对应的 PTE_DIRTY 标志位

```
// Overview:
// 把 bolckno 所对应的磁盘块标记为脏块，表示它对应的缓存页内容已经修改，需要重新写回磁盘
int dirty_block(u_int blockno) {
        void *va = disk_addr(blockno);

        if (!va_is_mapped(va)) {
                return -E_NOT_FOUND;
        }

        if (va_is_dirty(va)) {
                return 0;
        }
// 这里借助 syscall_mem_map 函数，重新设置 va 映射的页表中的标志位，显式增加写权限和被写过的脏位
        return syscall_mem_map(0, va, 0, va, PTE_D | PTE_DIRTY);
}
```

fs_init

函数功能：对文件系统进行初始化，初始化以后，super 和 bitmap 对应的磁盘块都被缓存到了文件管理进程中，之后再想访问二者，就不需要访问外存，直接在内存中访问即可

```c
// Overview:
// 初始化文件系统 : 读取超级块，检查磁盘是否可以正常工作，读取位示图
void fs_init(void) {
        read_super();
        check_write_block();
        read_bitmap();
}
```

check_write_block

函数功能：检查 write_block 这个函数是否可以正常工作（自检函数）

```c
// Overview:
// 函数功能 : 测试 write_block 是否可以正常工作，通过破坏超级块并对他重新读取进行验证
void check_write_block(void) {
        super = 0;

        // backup the super block.
        // copy the data in super block to the first block on the disk.
        panic_on(read_block(0, 0, 0));
        memcpy((char *)disk_addr(0), (char *)disk_addr(1), BLOCK_SIZE);

        // smash it
        strcpy((char *)disk_addr(1), "OOPS!\n");
        write_block(1);
        user_assert(block_is_mapped(1));

        // clear it out
        panic_on(syscall_mem_unmap(0, disk_addr(1)));
        user_assert(!block_is_mapped(1));

        // validate the data read from the disk.
        panic_on(read_block(1, 0, 0));
        user_assert(strcmp((char *)disk_addr(1), "OOPS!\n") == 0);

        // restore the super block.
        memcpy((char *)disk_addr(1), (char *)disk_addr(0), BLOCK_SIZE);
        write_block(1);
        super = (struct Super *)disk_addr(1);
}
```

以上就是所有和**磁盘块**有关的函数，fs.c 中剩下的函数都是以**文件**为读取单位，他们基于我们刚刚实现的**块读取**函数实现

文件控制块

struct File

在 MOS 中，描述文件使用文件控制块 **sruct File**，其定义于`user/include/fs.h`，每个控制块的大小为 256B，特冶被称为打开文件结构体，只有

文件控制块结构：

```
 struct File {
   char f_name[MAXNAMELEN];
   uint32_t f_size;
   uint32_t f_type;
   uint32_t f_direct[NDIRECT];
   uint32_t f_indirect;
   struct File *f_dir;
   char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 -
sizeof(void *)];
} __attribute__((aligned(4), packed));

#define BY2FILE 256
#define MAXNAMELEN 128
```

组成：

1. filename：文件名称，最大长度 MAXNAMELEN 为 128B
2. f_size：文件大小，以字节为单位
3. f_type：文件类型，分为普通文件（FTYPE_REG）和目录（FTYPE_DIR）两种
4. f_direct [N]：存放了 N 个磁盘块号的数组，每个元素都是一个磁盘块号，使用下标索引可以直接得到对应的磁盘块号
5. f_indirect：本身就是一个 u_int 类型的磁盘块号，通过这个磁盘块号所引到的磁盘块在内存中会被解释成一个 u_int 类型的数组，每个数组中又存着磁盘块号，这些磁盘块号对应的磁盘块真正用来和文件块一一对应
   `u_int *blk = read_block(f_indirect, blk, 0);`其中 blk 中就存放着一个个间接索引到的磁盘块号
6. f_dir：指向文件所属的文件目录，用来表示文件的位置
7. f_pad：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，确保一个 Block 能够包含整个文件块

三个结构体对比

struct File：文件控制块

struct Fd：文件描述符

struct Filefd：文件类型 Fd（包含 struct Fd 和 struct File）

结构体	位置	作用	应用层级
`struct File`	磁盘 & 内存	存储文件元数据 + 块指针	文件系统核心数据结构
`struct Fd`	用户内存	通用文件/设备描述符	用户态与内核通信桥梁
`struct Filefd`	用户内存	包含 `Fd` 和 `File`，描述打开的文件	用户态具体文件操作

目录项管理

在目录文件中，每个 File 结构体项就是一个**目录项**，可以近似认为二者是等价的，如果一个目录包含 20 个文件，那么这个目录文件的内容就是 20 个 File 结构体的顺序排列

目录文件、目录块、目录项的关系为：

若干个目录项组成目录块，若干个目录块组成目录文件，也就是目录文件中先是目录块，再是目录项

文件系统服务进程

与文件系统服务进程有关的函数存放在 fs / serv.c 中

struct Open

Open 结构体用于在文件系统服务端维护**打开文件**的状态，用于管理文件服务进程中所有的打开文件，这里可以把 Open 看成是文件服务的窗口，操作系统要想打开 / 操作文件，首先必须获得这个窗口，然后才能通过这个窗口对文件进行操作

```
struct Open {
  struct File *o_file; // 当前打开文件的文件控制块
  u_int o_fileid;      // 文件系统分配的唯一文件 ID，用来标识这个打开的文件
  int o_mode;          // 文件的打开模式
  struct Filefd *o_ff; // 映射给用户进程的文件描述符+文件控制块，允许用户访问，指向用户空间文件描述页的地址(也就是 fd 所在的地址，前面提过要给 fd 单独分配一段 PDMAP 存储)
};

struct Open opentag[MAXOPEN];// 管理所有 Open 标识符的数组
```

字段名	类型	描述
`o_file`	`struct File *`	指向磁盘上元数据结构，包含文件属性和内容映射信息
`o_fileid`	`u_int`	唯一文件标识符，用户系统调用通过它引用打开文件
`o_mode`	`int`	打开模式（只读/写/读写/追加等）
`o_ff`	`struct Filefd *`	指向用户空间文件描述页的地址，用于用户访问文件数据

我们在系统中设置了一个管理窗口的数组 opetab，通过遍历该数组，我们可以得到对应的 o_fileid，从而获得指定的窗口进行操作

serve_table

serve_table 是一个函数指针表，用于调度文件系统的服务请求，来自用户进程的 IPC 发送文件操作请求，根据请求号 req->req_type 选择对应的处理函数

```
void *serve_table[MAX_FSREQNO] = {
    [FSREQ_OPEN] = serve_open,   [FSREQ_MAP] = serve_map,
[FSREQ_SET_SIZE] = serve_set_size,
    [FSREQ_CLOSE] = serve_close, [FSREQ_DIRTY] = serve_dirty, [FSREQ_REMOVE]
= serve_remove,
    [FSREQ_SYNC] = serve_sync,
};
```

文件块 & 磁盘块 & 块缓存

每个磁盘块都要装入**固定**的块缓存中，但是这里指定的块号是**文件块号**，代表的是文件中的第 filebno 个块，而这个文件块在磁盘中具体是怎么存放的我们并不关心，因此可以直接通过 file_map_block 直接指定一个磁盘块号进行分配

更通俗的讲，可以把文件看成虚拟内存，而磁盘看成物理内存：

- 文件块是逻辑上连续的，每个文件都是如此
- 磁盘块是物理上连续的，同时，它在装入块缓存时又是一一对应的
- 相邻的文件块可以通过 `alloc_block` 函数**映射到**不相邻的磁盘块内

文件加载

其实我们要做的就是把文件加载到内存这件事，具体分成以下四个阶段：

文件逻辑块号 -> 磁盘块号 -> 虚拟内存页 -> 物理内存页

接下类我们会从这四个视角展示文件到物理内存之间的数据映射过程

文件视角

用户看到的文件结构对应文件的逻辑块号，以及相关的索引指针，这里模拟的是用户通过文件名访问文件的过程

```
File | filebno ... | f-direct | f-indirect
```

其中 filebno 是文件的逻辑块号，表示从文件角度看是第几个块，而 f-direct 和 f-indirect 分别表示直接块和间接块指针，用来指向磁盘中那些真正存储文件内容的**数据块**，通过调用 file_map_block() 函数可以访问

磁盘视角

磁盘所记录的都是具体的磁盘块号，由上层文件结构映射而来，每个 diskbno 对应磁盘上的一个物理块（但是连续的文件块对应的磁盘块可能是不连续的），而磁盘块号可以线性映射到进程空间 kseg1 的虚拟页中，也就是调用我们上面实现的 diskaddr() 函数实现

```
diskbno1 | diskbno2 | diskbno3
```

虚拟视角 & 物理视角

这部分映射和之前的内存管理一样，不涉及新的知识，需要借助页表完成虚拟地址和物理页面之间的映射

文件一般是存储在磁盘上的，只有在被访问到的时候才会读入物理内存中，而之所以不能直接把文件的逻辑块号映射到虚拟地址，是因为他们跨越了两个系统：文件系统和内存系统，必须借助磁盘块号和页缓存作为中间桥梁进行链接

文件操作函数 - fs.c

fs.c 中，在上文没有解析完的函数在这里，这些函数都会调用 block 级别的子函数

file_create

函数功能：创建 path 所指向的文件，返回文件控制块

```
// Overview:
/*
  函数功能 : 创建 path 路径所指定的文件，返回文件控制块 PCB，这里的返回值通过 file 指针来记录，函数返回值设为 0 表示成功
*/
int file_create(char *path, struct File **file) {
// name 表示路径中最后一级的文件名，r 表示错误码，dir 表示文件所在目录，f 表示目标文件(如果存在)或要新建的文件对象
```

```
        char name[MAXNAMELEN];
        int r;
        struct File *dir, *f;

// 调用 walk_path 函数找到路径中最后一集目录对应的 dir，查找该目录中是否有名为 name 的文件
        if ((r = walk_path(path, &dir, &f, name)) == 0) {
                return -E_FILE_EXISTS;
        }

        if (r != -E_NOT_FOUND || dir == 0) {
                return r;
        }

// 调用 dir_alloc_file 函数，在目录 dir 中分配一个空闲文件项 struct File
        if (dir_alloc_file(dir, &f) < 0) {
                return r;
        }

// 把从路径中提取出来的文件名 name 赋值到新文件结构中，然后把新创建的文件对象 f 返回给调用者
        strcpy(f->f_name, name);
        *file = f;
        return 0;
}
```

file_open

函数功能：直接调用了另一个函数 walk_path

```
// Overview:
/*
  函数功能 : 打开路径 path 所表示的文件
  如果打开成功，返回 0，并且让 file 指针指向被打开文件的文件控制块
*/
int file_open(char *path, struct File **file) {
        return walk_path(path, 0, file, 0);
}
```

walk_path

函数功能：从根目录开始依次解析文件名，搜索 path 所指向的文件，返回路径目录的控制块和文件控制块

```
// Overview:
/*
  函数功能 : 从根目录开始解析路径名称
  如果成功找到文件，就设置 pfile 指向文件，pdir 指向文件所在目录; 如果未找到文件，仅找到其所在目录，则设置 pdir 指向目录，并且把最后一级路径复制到 lastelem
*/
int walk_path(char *path, struct File **pdir, struct File **pfile, char *lastelem) {
// p 用于记录当前路径段的开始位置，name 用于记录当前路径段名称，dir 表示当前路径段所处的目录，file 表示查找到的文件或目录，r 保存返回自豪
        char *p;
        char name[MAXNAMELEN];
```

```
10          struct File *dir, *file;
11          int r;
12
13  // skip_slach 函数用来跳过 path 中的前导斜杠，也就是跳过根目录 /，从根目录 super->s_root 开始遍历，没有上一级目录，名称未空
14          path = skip_slash(path);
15          file = &super->s_root;
16          dir = 0;
17          name[0] = 0;
18
19  // 初始化输出参数指针
20          if (pdir) {
21                  *pdir = 0;
22          }
23
24          *pfile = 0;
25
26  // 只要路径还没解析结束，就按段处理路径
27          while (*path != '\0') {
28                  dir = file;
29                  p = path;
30
31                  while (*path != '/' && *path != '\0') {
32                          path++;
33                  }
34
35                  if (path - p >= MAXNAMELEN) {
36                          return -E_BAD_PATH;
37                  }
38  // 先保存当前路径的起始位置在 p 中，然后 path++ 直到找到下一个 / 或 path 路径的结尾，[p, path) 中就是下一段目录名
39                  memcpy(name, p, path - p);
40                  name[path - p] = '\0';
41                  path = skip_slash(path);
42                  if (dir->f_type != FTYPE_DIR) {
43                          return -E_NOT_FOUND;
44                  }
45  // 查找当前目录下，是否存在名为 name 的文件或子目录
46                  if ((r = dir_lookup(dir, name, &file)) < 0) {
47                    // 如果没找到目标名字，且路径已经遍历到最后一段，说明找到父目录，但是没找到文件本身，此时需要记录 pdir 和 lastelem
48                          if (r == -E_NOT_FOUND && *path == '\0') {
49                                  if (pdir) {
50                                          *pdir = dir;
51                                  }
52
53                                  if (lastelem) {
54                                          strcpy(lastelem, name);
55                                  }
56
57                                  *pfile = 0;
58                          }
59
60                          return r;
61                  }
62          }
63  // 反之，如果成功找到，就记录目标文件本身在 file 中
64          if (pdir) {
```

```
                *pdir = dir;
        }

        *pfile = file;
        return 0;
}
```

这个函数用于**解析路径字符串，从根目录开始逐层查找每一级目录或文件**：

- 如果能找到路径上所有的目录和文件，则返回最终文件结构体
- 如果找到的是路径上所有的目录，但最后一个文件不存在（比如用户打算创建新文件），会返回错误但保存好上一级目录和文件名，供调用者使用（如 `create()`）

file_get_block

函数功能：先获取文件中第 filebno 个逻辑块在磁盘中的块号（file_map_block <- file_walk_block），再把这个磁盘块中的数据从磁盘中**读到内存中**（read_block）

```
// Overview:
/*
  函数功能 : 把文件 f 中的第 filebno 个逻辑块对应的磁盘块中的数据加载到内存中，并且设置 blk 指向这段内存地址
  该函数会调用 file_map_block 和 read_block 两个函数
*/
int file_get_block(struct File *f, u_int filebno, void **blk) {
// r 临时存储函数调用返回值，diskbno 保存映射得到的磁盘块号，isnew 表示是否为第一次加载该磁盘块
        int r;
        u_int diskbno;
        u_int isnew;

// 通过文件逻辑块号 filebno 获取对应的磁盘块号 diskbno，设置 alloc = 1 表示需要自动分配磁盘块
        if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) {
                return r;
        }

// 调用 read_block 函数把磁盘块 diskbno 对应的内容加载到内存，并且返回这段内存的地址 blk
        if ((r = read_block(diskbno, blk, &isnew)) < 0) {
                return r;
        }
        return 0;
}
```

在函数调用完成后，`blk` 最终会**指向磁盘上 `diskbno` 所在块的内存映像**（即缓存中的一段内存），这一块就是文件中第 `filebno` 个逻辑块的数据，换言之，`*blk` 指向了磁盘块号为 `diskbno` 的内存地址，可以通过 `*blk` 来访问这个块的数据内容

file_map_block

函数功能：对 file_block_walk 的封装，获取第 filebno 块在磁盘中的块号；如果文件块 → 磁盘块的映射不存在，则会申请磁盘块形成一个映射

```
// Overview:
/*
```

```
  函数功能 : 把 diskbno 设置为文件 f 中第 filebno 个逻辑块所对应的磁盘块号，如果 alloc 为 1 且该磁盘块不存在，就分配一个新的磁盘块
*/
int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc) {
        int r;
        uint32_t *ptr;

// 查找文件逻辑块所对应的磁盘块指针，最后会在 ptr 中存储着 filebno 所对应的物理磁盘的块号
        if ((r = file_block_walk(f, filebno, &ptr, alloc)) < 0) {
                return r;
        }

// 如果磁盘块尚未分配，且允许分配，则分配新的磁盘块
        if (*ptr == 0) {
                if (alloc == 0) {
                        return -E_NOT_FOUND;
                }

                if ((r = alloc_block()) < 0) {
                        return r;
                }
                *ptr = r;
        }

// 把分配到的磁盘块号返回给调用者，存入 diskbno 指针中
        *diskbno = *ptr;
        return 0;
}
```

file_block_walk

函数功能：类似 pgdir_walk，获取文件 f 的第 filebno 块在磁盘中的磁盘块号；如果在非直接指针区域、没有间接指针块，会创建一个间接块（但是没有申请 filebno 的块）

```
// Overview:
/*
  函数功能 : 类似于 pgdir，只不过适用于文件系统
  查找文件 f 中第 filebno 个逻辑块对应的磁盘块号，并且将其地址存入 ppdiskbno 指针中
  如果 filebno 是直接块，则返回 f->f_direct[filebno] 对应的地址，如果是间接块，则访问间接块中相应的位置
  如果 alloc 值为 1，且需要分配间接块，则会自动分配间接块
  入参含义 :
    1. f 表示指向文件结构体的指针
    2. filebno 表示文件的逻辑块号(第几个块)
    3. ppdiskbno 用于存储查找到的磁盘块号的指针
    4. alloc 如果为 1，此时需要时可分配间接块
*/
int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int alloc) {
// r 临时保存返回值，ptr 指向找到的磁盘块号，blk 指向间接块数据
        int r;
        uint32_t *ptr;
        uint32_t *blk;

        if (filebno < NDIRECT) {
```

```
// 如果 filebno 小于直接块数量，说明我们访问的是直接块，直接获得对应的块号指针的地址即可
                ptr = &f->f_direct[filebno];
        } else if (filebno < NINDIRECT) {
// 如果是大于 NDIRECT 但是小于 NINDIRECT，说明我们访问的是间接块
                if (f->f_indirect == 0) {
                  // 如果还未分配间接块，根据需要进行分配，分配一个新的磁盘块作为间接块，并且记录块号到 f->f_indirect
                        if (alloc == 0) {
                                return -E_NOT_FOUND;
                        }

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

// 读取间接块内容(磁盘块号列表)到内存，用 blk 指向其内容，这里需要把间接块这个磁盘块本身解释成为一个 u_int 类型的数组，数组中每一个元素存放的都是一个磁盘块号
                if ((r = read_block(f->f_indirect, (void **)&blk, 0)) < 0) {
                        return r;
                }
            // 从间接块中取出第 filebno 个槽位，可以写成 &blk[filebno] 的形式，这里返回地址确实方便，如果想直接返回磁盘块号就写成 blk[filebno] 即可
                ptr = blk + filebno;
        } else {
                return -E_INVAL;
        }

// 把找到的磁盘块号的槽位地址传给 ppdiskbno，表示成功
        *ppdiskbno = ptr;
        return 0;
}
```

该函数实现了对文件 `f` 中第 `filebno` 个逻辑块的磁盘块号指针的定位：

- 如果是直接块，则直接返回对应的槽位指针；
- 如果是间接块：
  - 且未分配间接块时，可根据 `alloc` 判断是否分配；
  - 读取间接块内容，并返回槽位指针；
- 最终输出目标块号槽位地址到 `*ppdiskbno`，供调用者设置或读取

file_set_size

函数功能：设置指定文件的大小，会调用 file_clear_block，file_truncate，file_flush 这三个底层函数

```
// Overview:
// 函数功能 : 设置文件 f 的大小为指定的 newsize
int file_set_size(struct File *f, u_int newsize) {
        if (f->f_size > newsize) {
                file_truncate(f, newsize);
        }

        f->f_size = newsize;

        if (f->f_dir) {
```

```
                file_flush(f->f_dir);
        }

        return 0;
}
```

在该函数内部，如果文件变小，需要释放掉曾经和现在相比多占用的那部分空间（truncate），同时要把修改后的文件内容从内存写回到磁盘（flush）

file_clear_block

函数功能：释放某个磁盘块的空间

```
// Overview:
// 函数功能 : 从文件 f 中移除编号为 filebno 的数据块，如果该块本身就不存在，默认移除成功
int file_clear_block(struct File *f, u_int filebno) {
        int r;
        uint32_t *ptr;

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

        if (*ptr) {
                free_block(*ptr);
                *ptr = 0;
        }

        return 0;
}
```

file_truncate

函数功能：释放文件缩小后曾经占用但现在不需要的那些多余的块

```
// Overview:
/*
  函数功能 : 把文件大小截断为 newsize 字节，并且释放掉曾经占用的，现在不需要的那些磁盘块对应的空间
  对于旧文件大小和新文件大小，分别计算其所需要的块数，然后从 new_nblocks 到 old_nblocks 的块需要全部清除
  如果 new_nblocks 不超过 NDIRECT，说明间接块也不需要了，也要释放，把 f->f_indirect 清除
*/
void file_truncate(struct File *f, u_int newsize) {
        u_int bno, old_nblocks, new_nblocks;

        old_nblocks = ROUND(f->f_size, BLOCK_SIZE) / BLOCK_SIZE;
        new_nblocks = ROUND(newsize, BLOCK_SIZE) / BLOCK_SIZE;

        if (newsize == 0) {
                new_nblocks = 0;
        }

        if (new_nblocks <= NDIRECT) {
                for (bno = new_nblocks; bno < old_nblocks; bno++) {
```

```
                        panic_on(file_clear_block(f, bno));
                }
                if (f->f_indirect) {
                        free_block(f->f_indirect);
                        f->f_indirect = 0;
                }
        } else {
                for (bno = new_nblocks; bno < old_nblocks; bno++) {
                        panic_on(file_clear_block(f, bno));
                }
        }
        f->f_size = newsize;
}
```

file_flush

函数功能：同步文件内容至磁盘，此处用于更新被修改文件的目录，也就是同步目录中的所有文件

```
// Overview:
/*
  函数功能 : 把文件 f 的内容刷新到磁盘上
  遍历文件中所有逻辑块，把逻辑块号转化为磁盘块号，然后检查该磁盘块是否设置了 dirty 位标志，如果是，就需要把它写回磁盘
*/
void file_flush(struct File *f) {
        u_int nblocks;
        u_int bno;
        u_int diskbno;
        int r;

        nblocks = ROUND(f->f_size, BLOCK_SIZE) / BLOCK_SIZE;

        for (bno = 0; bno < nblocks; bno++) {
                if ((r = file_map_block(f, bno, &diskbno, 0)) < 0) {
                        continue;
                }
                if (block_is_dirty(diskbno)) {
                        write_block(diskbno);
                }
        }
}
```

flie_close

函数功能：关闭文件

```
// Overview:
// 函数功能 : 关闭一个文件
void file_close(struct File *f) {
// 调用 file_flush 函数，把文件 f 当前在内存中修改过的内容写回磁盘，确保数据同步，这是关闭文件前的必要操作
        file_flush(f);
        if (f->f_dir) {
                u_int nblock = f->f_dir->f_size / BLOCK_SIZE;
                for (int i = 0; i < nblock; i++) {
                        u_int diskbno;
```

```
                        struct File *files;
                        if (file_map_block(f->f_dir, i, &diskbno, 0) < 0) {
                                debugf("file_close: file_map_block
failed\n");
                                break;
                        }
                        if (read_block(diskbno, (void **)&files, 0) < 0) {
                                debugf("file_close: read_block failed\n");
                                break;
                        }
                        if (files <= f && f < files + FILE2BLK) {
                                dirty_block(diskbno);
                                break;
                        }
                }
                file_flush(f->f_dir);
        }
}
```

file_dirty

函数功能：调用前面实现的函数 dirty_block，实现对指定文件的**特定地址**设置 DIRTY 标识（在写入 offset 以后，为确保同步块缓存和磁盘数据，需要调用这个函数）

```
// Overview:
// 函数功能 : 标记文件 f 中偏移量为 offset 所在的块(也就是第 offset/BLOCK_SIZE 号块)
为 dirty 块，offset 是页对齐的
int file_dirty(struct File *f, u_int offset) {
        int r;
        u_int diskbno;

        if ((r = file_map_block(f, offset / BLOCK_SIZE, &diskbno, 0)) < 0) {
                return r;
        }

        return dirty_block(diskbno);
}
```

file_remove

函数功能：通过字符串展示的路径先查询到文件控制块，然后使用将自身大小减为 0 的方式，解除所有占用的内存块，最后将修改同步到磁盘内

```
// Overview:
// 通过截断文件内容，并且清空文件名，实现文件删除，给定 path，根据 path 路径删除文件
int file_remove(char *path) {
        int r;
        struct File *f;

// 通过 path 路径查找对应的 File 结构体
        if ((r = walk_path(path, 0, &f, 0)) < 0) {
                return r;
        }

// 把文件内容长度截断为 0，相当于清空文件的数据(但此时文件还存在，只是内容为空)
        file_truncate(f, 0);
```

```
14
15  // 清空文件名，把 File 结构体中的 f_name 字段设置为 0，这意味着文件目录中这个条目消失，从目录列表中删除该文件的引用
16          f->f_name[0] = '\0';
17
18  // 写回修改，把文件写回磁盘(其实这里文件已经清空，写回磁盘的就是一个空文件)，保证数据持久化，如果文件目录非空，目录本身内容也要写回磁盘，保证目录结构同步更新
19          file_flush(f);
20          if (f->f_dir) {
21                  file_flush(f->f_dir);
22          }
23
24          return 0;
25  }
```

file_sync

函数功能：将**整个文件系统中所有被修改过（即 dirty）的磁盘块**都**写回磁盘**，确保内存中的所有更新都持久化到磁盘上，避免数据丢失

```
1   // Overview:
2   // 函数功能 : 同步整个文件系统的所有数据到磁盘，是一种强制的彻底的同步操作，会检查所有磁盘块(所有的文件，目录，元数据等)，只要标记了 dirty 位，都要全部写回磁盘
3   void fs_sync(void) {
4           int i;
5           for (i = 0; i < super->s_nblocks; i++) {
6             // 文件系统中所有的磁盘块编号为 [0, super->w_nblocks - 1]
7                   if (block_is_dirty(i)) {
8                           write_block(i);
9                   }
10          }
11  }
```

dir_lookup

函数功能：在 dir 目录中，查找名称为 name 的文件，具体实现方式为遍历目录中的每个文件，查找名称是否相同；同时，遍历的方式是先块后文件

```
1   // Overview:
2   /*
3     函数功能 : 在目录 dir 中查找名为 name 的文件，如果找到，把其文件控制这块写入 file 指针中，这是一个典型的目录遍历函数
4     入参含义 :
5       1. dir 为文件目录指针，该目录中包含若干个 File 结构体
6       2. name 为要查找的文件名
7       3. nblock 表示 dir 所占用的数据块数量
8       4. blk 表示指向某个目录块的缓冲区指针
9       5. FILE2BLK 表示一个磁盘块中容纳的 File 结构体的数量(宏定义)
10  */
11  int dir_lookup(struct File *dir, char *name, struct File **file) {
12  // 计算目录文件 dir 实际占用了多少个数据块
13          u_int nblock;
14          nblock = dir->f_size / BLOCK_SIZE;
15
16  // 遍历目录文件中的每一个块
17          for (int i = 0; i < nblock; i++) {
```

```
// 获取目录中的第 i 个块在内存中的映射地址 blk，如果块还未加载，则会从磁盘中读取
                void *blk;
                try(file_get_block(dir, i, &blk));
                struct File *files = (struct File *)blk;
// 把该块的内容解释成一个 File 数组，说明一个目录文件中的每个块存储的是一组文件项(每个项是一个 struct File)，下面遍历这个块中所有的 File 结构体
                for (struct File *f = files; f < files + FILE2BLK; ++f) {
// 比较当前目录项的名字和目标名字 name 是否相同，f->f_name 是当前子文件或子目录名称，如果找到目标文件，先记录其文件控制块，再设置文件所属目录，直接返回 0
                        if (strcmp(name, f->f_name) == 0) {
                                *file = f;
                                f->f_dir = dir;
                                return 0;
                        }
                }
        }

        return -E_NOT_FOUND;
}
```

dir_alloc_file

该函数的作用是：在一个目录文件 `dir` 中寻找一个空闲的 `File` 结构（即一个目录项），用来创建一个新文件或子目录。如果原有的目录块都没有空闲空间，则会为目录添加一个新的数据块

该函数是文件系统中创建文件 / 目录的关键步骤之一

```
// Overview:
// 在指定目录下分配一个新的 File 结构体，在目录 dir 中找到一个空闲的 File 结构，把其地址赋值给 file 指针
int dir_alloc_file(struct File *dir, struct File **file) {
        int r;
        u_int nblock, i, j;
        void *blk;
        struct File *f;

        nblock = dir->f_size / BLOCK_SIZE;
// 两重循环遍历，先遍历块，再遍历块内的文件项
        for (i = 0; i < nblock; i++) {
// 把第 i 个数据块映射到内存中，内存地址记录在 blk 中
                if ((r = file_get_block(dir, i, &blk)) < 0) {
                        return r;
                }
// 把每块内容解释成为一个 struct File 数组，每个目录块中保存的是一组目录项(每项都是一个 File 结构)
                f = blk;
// 遍历这个块中的所有 File 项，寻找空闲的项，通过检查1f_name[0] == '\0' 来判断
                for (j = 0; j < FILE2BLK; j++) {
                        if (f[j].f_name[0] == '\0') {
                                *file = &f[j];
                                return 0;
                        }
                }
        }

// 如果目录块没有空位，就需要为目录扩展一个新的数据块，增加 f_size 相当于申请新块的型号，此时 i == nblock，重新获得新的一块，且新块是空的，返回第一个目录项地址作为空闲项
```

```
        dir->f_size += BLOCK_SIZE;
        if ((r = file_get_block(dir, i, &blk)) < 0) {
                return r;
        }
        f = blk;
        *file = &f[0];

        return 0;
}
```

文件进程服务函数

和文件进程服务有关的函数存在 serv.c 中

在 serv.c 和 fsipc.c 中我们使用到进程间的通信函数，也就是 ipc_send 和 ipc_recv，这里我们先回顾二者的参数含义

```
// 发送函数 : whom 表示消息接收者的进程 id，val 表示发送的数值消息，srcva 表示发送方想传递给对方的数据的起始地址，perm 表示对应页面的权限位
void ipc_send(u_int whom, u_int val, const void *srcva, u_int perm);
// 接收函数 : whom 表示消息发送者的进程 id，dstva 表示接收方用于接收对方传输数据的起始内存地址，perm 表示对应页面的权限位
u_int ipc_recv(u_int *whom, void *dstva, u_int *perm);
```

serve_open

函数功能：根据输入，申请一个工作区后打开文件并且**保存**文件信息，完成后直接调用 ipc_send 返回

```
/*
 * Overview:
   函数功能 : 打开一个由 rq 中的路径指定的文件，她会尝试分配一个文件描述符，打开该文件，如果打开成功，把打开的信息保存在该文件描述符中，并且通过 ipc_send 把 FileFd 页返回给调用者
   入参含义 :
     1. envid 表示请求进程的环境 ID
     2. rq 表示请求结构体，包含路径和打开模式
*/
void serve_open(u_int envid, struct Fsreq_open *rq) {
// f 指向文件控制块(文件目录项)，ff 指向用户空间返回的文件描述符页，r 临时记录返回值，o 表示内核中打开的文件结构 Open 表项
        struct File *f;
        struct Filefd *ff;
        int r;
        struct Open *o;

// 找到一个空闲的 Open 表项，open_alloc 已在打开文件表中找到一个空闲项，并且返回指针 o
        if ((r = open_alloc(&o)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

 // 如果设置了 O_CREAT 标志但文件不存在，就需要用 f_create 创建文件，O_CREAT 是一个标识符，表示若文件不存在就需要创建
        if ((rq->req_omode & O_CREAT) && (r = file_create(rq->req_path, &f)) < 0 &&
            r != -E_FILE_EXISTS) {
                ipc_send(envid, r, 0, 0);
```

```
                return;
        }

// 无论创建是否通过，调用 file_open 根据路径加载文件元数据结构，也就是 struct File
        if ((r = file_open(rq->req_path, &f)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

// 把 struct File 指针保存到打开表 Open 中，表示该打开文件对应的实际文件
        o->o_file = f;

// 如果设置了 O_TRUNC，需要截断文件，清空内容，O_TRUNC 为标识符，表示打开文件时清空文件的内容
        if (rq->req_omode & O_TRUNC) {
                if ((r = file_set_size(f, 0)) < 0) {
                        ipc_send(envid, r, 0, 0);
                }
        }

// 填充 FileId(用户文件描述符页)
    ff = (struct Filefd *)o->o_ff;
    ff->f_file = *f;                        // 拷贝文件元数据到用户空间
    ff->f_fileid = o->o_fileid;             // 绑定一个唯一 ID
    o->o_mode = rq->req_omode;             // 保存打开模式
    ff->f_fd.fd_omode = o->o_mode;          // 用户态也需要知道打开模式
    ff->f_fd.fd_dev_id = devfile.dev_id;    // 指明该文件来自哪个设备（文件系统）

// 使用 ipc_send 把含有文件信息的页通过进程间通信返回给 envid 对应的进程
        ipc_send(envid, 0, o->o_ff, PTE_D | PTE_LIBRARY);
}
```

用户态定义与接口

在之前的分析中，我们已经完成了文件管理进程中的函数实现。现在我们来分析用户态中用户直接可用的函数接口与数据结构定义

我们还是从距离文件系统**最近**的函数与文件开始：`user/lib/fsipc.c` 和 `user/include/fsreq.h`

首先明确用户进程和文件管理进程**使用 IPC 通信进行交互**，在文件管理进程的运行的核心、分发函数 `serve` 中，我们通过 `ipc_recv` 从用户进程获取到了一片虚拟地址 `REQVA`，并且针对不同的信息，将 `REQVA` 转换成了不同的数据结构，进行处理，这些数据结构就定义在 `fsreq.h` 文件中

文件描述符 fd 是系统给用户提供的整数，使得用户可以在描述符表（单独占一个页面）中进行索引

用户在使用 I/O 编程的时候，使用 open 在描述符表的指定位置存放被打开文件的信息，使用 close 把描述符表中指定位置的文件信息释放，使用 write 和 read 修改描述符表指定位置处的文件信息（**也就是说**，用户是看不到他所操作的文件的实体的，它只能看到它操作的文件的**相关信息**，而这些信息就借助文件描述符来存储文件的基本信息和用户进程中有关文件的状态，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，文件系统进程会把这些基本信息记录在内存（全局打开文件表 opentab）中，然后由操作系统把用户进程请求的地址映射到一个存储了文件描述符的物理页（filebase）上；当用户进程获得了文件大小等基本信息后，再次向文件系统发送请求，文件内容又会被映射到指定的内存空间（文件缓存）中

fsreq.h

该文件中的结构体和宏定义，都是在操作系统中实现用户态文件系统服务的 IPC 接口的定义，他们用来定义不同类型的文件系统请求编号（enum）和对应请求的数据结构（struct Fsreq_*），用于实现用户进程和文件系统服务进程之间的通信

enum

文件系统请求号：为每种文件系统请求定义一个唯一的编号

枚举常量	请求含义	说明
`FSREQ_OPEN`	打开文件	使用 `Fsreq_open` 结构体
`FSREQ_MAP`	将文件内容映射到进程地址空间	使用 `Fsreq_map` 结构体
`FSREQ_SET_SIZE`	设置文件大小	使用 `Fsreq_set_size`
`FSREQ_CLOSE`	关闭文件	使用 `Fsreq_close`
`FSREQ_DIRTY`	标记某个页为脏页（修改过）	使用 `Fsreq_dirty`
`FSREQ_REMOVE`	删除文件	使用 `Fsreq_remove`
`FSREQ_SYNC`	同步所有数据到磁盘	一般不带参数
`MAX_FSREQNO`	请求类型数量上限（用于边界检查）	——

文件请求服务函数

和用户态有关的函数都定义在 fsipc.c 这个文件中，而这个文件中最重要的函数就是其**承上启下**作用的 fsipc 函数，其他函数只是完成了对应请求的**参数填写**，并且最终调用这个函数来实现的

fsipc

file system IPC：`fsipc()` 把请求类型（如 `FSREQ_REMOVE`、`FSREQ_OPEN` 等）和请求数据（如 `fsipcbuf`）发给文件服务器进程，然后等待服务器返回数据（可选的返回页）和返回值

```
// Overview:
/*
  函数功能 : 向文件系统服务器发送一个 IPC 请求，行窃等待服务器的响应
  入参含义 :
    1. type 表示请求类型(比如 TSREQ_OPEN，FSREQ_REMOVE)，作为简单整数值由 IPC 发送
    2. fsreq 表示请求数据所在的页，通常是 fsipcbuf(共享内存)
    3. 如果用户进程期望接收文件系统返回的一页数据，传入虚拟地址；否则传 0
    4. perm 表示接收到的页的权限位
*/
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm) {
        u_int whom;
// envs[1].env_id : 接收进程 ID(文件系统服务器); type : 发送的整型请求号(比如 FSREQ_OPEN);  fsreq : 携带请求参数的数据页(如 fsipcbuf); PTE_D : 发送页的权限(可写 | 用户 | 脏页)
        ipc_send(envs[1].env_id, type, fsreq, PTE_D);
// &whom : 输出参数，接收到的消息来自哪个进程; dstva : 接收页的虚拟地址映射位置(如果服务器返回了某个页面); perm : 输出参数，记录返回页的权限
        return ipc_recv(&whom, dstva, perm);
}
```

这里规定在 MOS 系统中，文件系统进程必须为第二个进程 envs[1]，保证传输的正确性

在每一个用户态的每一个请求服务函数中，也就是以 fsipc_ 作为前缀的函数，都会用到一个临时的页面用来传输数据：

```
u_char fsipcbuf[BY2PG] __attribute__((aligned(BY2PG)));
```

fsipcbuf 用于在文件系统客户端和文件系统服务端之间共享数据的一页内存，其中长度为 BY2PG，说明大小是一页，u_char 表示这块内存是按字节访问的

通过修改页面内的数据，搭配不同的服务函数，让文件管理进程以不同的方式去解析这片地址空间，**以实现传输相同页面却能实现不同功能的效果**

fsipc_remove

函数功能：在用户态下请求删除 path 路径所对应的文件，注意这里的路径删除是 path 路径所指向的**最后一级**文件，而不是整个路径上的所有内容

```
// Overview:
// 函数功能 : 向文件服务器请求删除一个给定路径的文件
int fsipc_remove(const char *path) {
// 使用 strlen 检查 path 的长度，如果 path 长度为 0 或者超过长度上限，则直接报错
        if (path[0] == '\0' || strlen(path) >= MAXPATHLEN) {
                return -E_BAD_PATH;
        }

// 把 fsipcbuf 作为 struct Fsreq_remove 使用(这就上文所说的，不同用户请求函数会对 fsipcbuf 这一个页面的空间做不同的解释)，还是使用结构体间的类型强转来实现
        struct Fsreq_remove *req = (struct Fsreq_remove *)fsipcbuf;

// 使用 strcpy 把 path 拷贝到 req 中的路径子段
        strcpy((char *)req->req_path, path);

// 使用 fsipc 向服务器发出删除请求，fsipc 的参数列表如下
    /*
      1. FSREQ_REMOVE : 操作类型的标识符
      2. req : 请求结构体，含有这个请求操作的参数
      3. 后两个参数表示不需要文件系统服务器返回数据缓冲区的虚拟地址，因此也不需要设置标志位 perm
    */
        return fsipc(FSREQ_REMOVE, req, 0, 0);
}
```

纯文件操作函数

再往上一层，调用 fsipc.c 这个文件中的那些用户态的**请求服务函数** `fsipc_*.c` 的是位于 `user/lib/file.c` 的一系列函数，他们是用户态中可以直接执行文件操作的函数，但他们还不是最顶层的函数

（可以类比系统调用中调用的 syscall_* 一级的函数，他们向下会调用 msyscall，但向上还会被用户态的顶层接口所调用）

open

```
// Overview:
/*
  函数功能 : 打开一个文件或目录，成功时返回文件描述符
  入参含义 : path 表示待打开文件的路径，mode 表示打开方式
*/
int open(const char *path, int mode) {
        int r;

// 分配一个文件描述符 struct Fd(Fd 用来记录文件的元信息，缓冲区，偏移量等)
        struct Fd *fd;
        r = fd_alloc(&fd);
        if (r) {
                return r;
        }

// 请求文件服务器打开文件，通过调用 fsipc_open 实现
        r = fsipc_open(path, mode, fd);
        if (r) {
                return r;
        }

// 读取文件信息，获取文件大小和文件 ID
/*
  va 表示该文件所在用户地址空间中缓冲区的起始地址(fd->data 区域)
  ffd 为 fd 强制转化为 struct Filefd* 类型，包含具体的文件信息
  size 为文件的大小
  fileid 为文件服务器为该文件唯一分配的 ID，用于后续 IPC 请求
*/
        char *va;
        struct Filefd *ffd;
        u_int size, fileid;
        va = fd2data(fd);
        ffd = (struct Filefd *)fd;
        size = ffd->f_file.f_size;
        fileid = ffd->f_fileid;

// 把文件内容映射到用户地址空间，每次把一个大小为 PTMAP 的文件也从文件服务器映射到用户的虚拟地址空间 va + i
        for (int i = 0; i < size; i += PTMAP) {
                r = fsipc_map(fileid, i, va + i);
                if (r) {
                        return r;
                }
        }

// 把 fd 转化为整数类型的文件描述符编号(fd 表的下标)
        return fd2num(fd);
}
```

remove

```
// Overview:
// 函数功能 : 删除一个文件或目录，成功时返回 0
int remove(const char *path) {
// 直接调用 fsipc_remove，通过 IPC 机制向文件服务器发送删除请求即可
        return fsipc_remove(path);
}
```

文件系统顶层函数

文件系统中真正能由用户态所直接调用的顶层函数存储在 fd.c 文件中

struct Dev

文件多样性

操作系统中支持的“文件操作”其实不局限于“真实文件”，还包括：

1. **文件设备**（Disk File）——例如硬盘上的文件，最常见
2. **控制台**（Console）——例如终端、键盘输入/输出
3. **管道**（Pipe）——进程之间通信用的管道

只不过在 lab5 中，我们只需要考虑真正的文件设备的操作，对于后两种暂时不需要考虑，而涉及到对这些真实文件操作的函数主要在 `file.c` 中实现，比如 `open()`、`read()`、`write()`、`close()`，这些函数通过 IPC 和文件系统服务器打交道，属于文件类型的设备处理

设备分发机制

在用户或系统调用层，我们通常只看到 `read(fd, ...)`、`write(fd, ...)`，但这些函数并不知道 `fd` 指向的到底是文件、控制台还是管道，因此，系统使用一个 `fd->dev_id` 来标识这个文件描述符对应的是哪种设备，而在每个函数内部会根据不同的设备类型进行分发，让每个设备对应一个执行功能的函数，再把分发函数封装在执行功能的函数内部

比如在 fd.c 的 read 函数中可能会有以下代码：

```
Dev *d = dev_lookup(fd->fd_dev_id);
return (*d->dev_read)(fd, buf, n);
```

其中 dev_lookup 类似于一个查表函数，可以根据 fd_dev_id 字段找到该文件描述符所属的设备类型，并且返回该设备类型对应的 Dev 结构体指针，待查找的设备表按数组形式组织，在查找过程中会逐项比对每一个条目：

```
Dev devtab[] = {
    [DEV_FILE] = { .dev_read = file_read, .dev_write = file_write, ... },
    [DEV_CONSOLE] = { .dev_read = cons_read, .dev_write = cons_write, ... },
    [DEV_PIPE] = { .dev_read = pipe_read, .dev_write = pipe_write, ... },
};
```

而 return 一句则是利用了**函数指针嗲用**，调用设备 d 的 dev_read，传入当前的 fd，目标缓冲区 bif，以及读取的字节数 n，其中根据设备类型 d 的不同，d 对应的 read 函数指针 d->dev_read 所指向的函数体也不同，比如

1. 如果设备是文件，则 `d->dev_read` 指向的是 `file_read()` 函数
2. 如果设备是控制台，则 `d->dev_read` 指向的是 `cons_read()` 函数

3. 如果设备是管道，则 `d->dev_read` 指向的是 `pipe_read()` 函数

所以最后一句 return 其实等价于一个分支条件：

```
file_read(fd, buf, n);       // 如果是文件
cons_read(fd, buf, n);       // 如果是控制台
pipe_read(fd, buf, n);       // 如果是管道
```

struct Dev

总体来讲，上文提到的三类"文件"设备平等，并且在文件系统中视角来看，都属于 `Fd` ，即 file descriptor 的一种，每类设备拥有自己的**功能函数**以处理对应的请求，并且通过各自的函数指针索引到对应的功能函数，三种设备的信息统一存放在一个 `Dev` 数组内，设备具体又分别定义在各自功能函数的头文件内

Dev 结构体的组织如下：

```
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);
};
```

该结构体中即存储了各个设备对应的文件操作函数的指针

sruct Fd

Fd 就是我们所说的**文件描述符**，每一个用户进程可以同时操控多个设备，为了在用户态对这些设备加以区分，就引入了文件描述符的定义，通过访问对应设备id文件描述符，就可以得知它的设备种类（dev_lookup），因此我们只需要传入文件描述符，就能够自动获得 Dev 中存放的函数指针并执行

注意，每个进程的文件描述符的上限为 32 个，且每个描述符会**单独**占用一个**虚拟页**，还对应一个 PDMAP 大小的 **data** 空间处理该设备打开后处理的内容，需要**注意**的是，Fd 结构体和 data 缓冲区所占的两个页都是**逻辑上**的概念，只会在进程虚拟地址中为其划分地址，只有真正操作文件的时候，他们才有可能被加载仅内存中

文件描述符 fd（file descriptor）是**用户程序**管理、操作文件的基础，其结构组织如下：

```
struct Fd {
  u_int fd_dev_id;  // 此设备对应的 Dev 类型
  u_int fd_offset;  // 目前的文件指针距离起始的偏移量
  u_int fd_omode; // 设备的打开模式
};
```

操作系统用户态文件描述符（file descriptor ）与其背后内核支持结构的内存映射机制如下：

当我们打开一个文件的时候，得到的文件描述符（函数返回值）是一个整数（比如 fd == 3），这其实是一个**下标**，是当前文件的文件描述符在整个文件描述符 struct Fd 数组中的下标，我们需要相关的**辅助函数**进行翻译

1. struct Fd* fd_lookup(int fd)：把 fd 下标变成 Fd 结构体指针，找到对应的文件描述符结构体
2. char* fd2date(struct Fd *fd)：根据下标 fd 找到文件的数据缓冲区 data

而我们会在**用户进程的虚拟空间**中，按顺序存储的方式依次为每个 fd 和 data 段分配虚拟页面

```
        0x60000000 → FILE.BASE
          |
          |-- data[0] (for fd 0)
          |-- data[1] (for fd 1)
          |-- data[2] (for fd 2)
          ...
          |
        FDTABLE → Fd[0] (at FDTABLE + 0*PGSIZE)
                → Fd[1] (at FDTABLE + 1*PGSIZE)
                → Fd[2] ...
```

总结：用户程序通过整数型的文件描述符（fd）来访问文件，而操作系统通过将这些整数映射到一片预先分配的虚拟内存区域中（FDTABLE + FILE.BASE），并借助辅助函数（如 fd_lookup、fd2data）实现了从 fd 到其结构体元数据和数据区的定位与访问

struct Filefd

为了更加容易实现对纯文本的操作，再次对 Fd 结构体进行了封装，实现了一个专门为 file 使用的 FileFd 结构，它包含了更多信息，能够更容易地进行 file 类的操作

```
struct Filefd {
  struct Fd f_fd; // 文件描述符的通用部分(设备、偏移、打开模式)
  u_int f_fileid; // 文件 ID
  struct File f_file; // 具体文件元数据(文件名，类型，大小，直接块，间接块等)
};
```

实际上我们可以把 Filefd 结构体就当成是 Fd 结构体来使用，因为前者的前三个字段和后者的前三个字段都是 u_int 类型，这里涉及到结构体的类型继承

结构体类型继承与强转技巧

对于类型为 Fd 的一段内存，该段内存开头被解释为三个 u_int 类型的数据，将其强制转换为 Filefd 类型之后，由于 Filefd 第一个成员是 Fd 类型，对这段内存开始部分的解释依旧可以是三个 u_int 类型的数据（通过 f_fd 成员即可访问到），同时如果是 Filefd 类型我们还可以继续向后访问，第二个成员开始的内存将会按照 FileFd 的定义进行解释（因为能转成 Filefd，则说明后面的字段都有意义）

若将 Filefd 类型强制转换为 Fd 将会遮盖掉不同细分类别的文件描述符的不同，也就是只保留前三个 u_int 类型的通用字段，后面特定文件的元信息就会被隐藏

相当于实现了一个支持**多态**的结构体：

目的	方法	结果
支持“多态”文件描述符接口	将具体结构的第一个字段设为通用结构（如 `Fd`）	可以用 `Fd*` 统一操作所有 `Filefd*`
从通用类型转换到具体类型	强制将 `Fd*` 转换为 `Filefd*`	可访问更丰富的特定字段
从具体类型转换为通用类型	强制将 `Filefd*` 转换为 `Fd*`	实现统一操作，隐藏细节

我们之前提到，`Open` 结构体类似于一个文件服务进程使用的“窗口”，实际上文件描述符 `Fd` 也**类似一个在用户态中的窗口**，只不过它的范围更广，可以包含除了 `File` 以外的其他设备，但同时它也只为自己的进程而服务

struct Stat

用来表示某个文件的状态

```
// 文件状态结构体
struct Stat {
        char st_name[MAXNAMELEN]; // 文件或目录名
        u_int st_size;  // 文件大小
        u_int st_isdir; // 是否为目录
        struct Dev *st_dev; // 所属设备(指向设备类型结构体)
};
```

Stat 结构体主要是为了用户能够快速查看文件状态所设计的，在 fstat 和 stat 链中最终被访问

用户访问流程

用户态根据 Fd 执行功能函数，具体分为以下几步：

1. 根据 Fd 号找到对应的 fd 数据
2. 判断 Dev 种类，获取执行操作的函数指针
3. 直接调用函数指针，实现该 Dev 自行实现的底层函数，完成操作

我们以用户态顶层的 read 函数为例分析上述过程，首先用户态会执行 fd.c / read 函数：

```
// Overview:
/*
  函数功能 : 从文件描述符 fd 当前的偏移位置开始，读取最多 n 个字节到 buf 缓冲区中，如果读取成功，更新文件的偏移量(下次开始读写的位置)，返回成功读取的字节数
  入参含义 :
    1. fdnum 为文件描述符编号
    2. buf 为读取数据要存放的缓冲区地址
    3. n 为要读取的最大字节数
*/
int read(int fdnum, void *buf, u_int n) {
        int r;
// fd 为指向文件描述符(struct Fd)的指针，表示当前打开的文件或设备，dev 为指向当前设备描述符(struct Dev)的指针，描述一个抽象的设备类型
        struct Dev *dev;
        struct Fd *fd;
// 先把文件描述符编号 fdnum 映射成文件描述符结构体 fd，再通过文件描述符中的 fd_dev_id 找到对应设备的函数集，确定 fd 是何种类型的设备
        if ((r = fd_lookup(fdnum, &fd)) < 0) {
                return r;
        }
        if ((r = dev_lookup(fd->fd_dev_id, &dev)) < 0) {
                return r;
        }

// fd_omode 表示文开打件模式，O_ACCMODE 是提取访问模式位的掩码，如果提取出来的打开模式是 O_WRONLY，则直接报错
/*
  1. O_RDONLY : read only
```

```
  2. O_WRONLY : write only
  3. O_RDWR : both can write and read
*/
        if ((fd->fd_omode & O_ACCMODE) == O_WRONLY) {
                return -E_INVAL;
        }

// 通过函数指针调用设备的读函数，传入当前文件描述符，缓冲区，读取的长度和偏移量，dev_read 是一个函数指针，来自 dev 中的函数集，它最终会调用具体设备的 read 实现

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

// 如果读取成功，就更新偏移量，下次会从本次读取末尾的下一个字符开始读取，并且返回读取到的字节数
        if (r > 0) {
                fd->fd_offset += r;
        }

        return r;
}
```

所以实际上只需要准备好 dev_* 函数即可，而因为此时我们只会操作文件设备，不涉及管道等，所以可以直接进行如下的初始化：

```
// 点号（.）表示使用结构体中的成员名来初始化
// 而不需要按照结构体中声明成员的顺序来依次赋值
struct Dev devfile = {
    .dev_id = 'f',              // 'f' 通常表示 "file" 类型的设备
    .dev_name = "file",         // 设备名为 "file"，用于识别
    .dev_read = file_read,      // 指向文件读操作的函数
    .dev_write = file_write,    // 指向文件写操作的函数
    .dev_close = file_close,    // 指向文件关闭函数
    .dev_stat = file_stat,      // 指向文件状态查询函数
};
```

本质上是注册了一个**文件设备**的操作集，当一个设备的 ID 是 ‘f’ 的时候，就应该使用这个设备集中的函数进行相关操作

文件缓存

在上面的设计中，我们进行了两次映射：

1. 把文件的数据块 data 段映射到 FILEBASE 段的虚拟地址，作为文件缓存
2. 把磁盘块映射到 DISKMAP 段的虚拟地址，作为块缓存

这两部分是不矛盾的，且两段地址空间在进程内核空间中是不重叠的，前者可以看成是 cache，直接访问缓存中的文件数据比访问磁盘块要快得多

当我们初次访问某个文件内容的时候，因为曾经没有访问记录，必须去文件所在的磁盘块，进行按块访问；访问完以后，就需要把对应的文件块调入文件对应的缓存中，之后再访问这个文件块，就可以去它的**缓存**处访问

注意

在 FILEBASE 段不仅为文件的 data 预留了缓存，也为 fd 结构体预留了缓存，后者存储的空间称为 FDTABLE，因此需要再 FILEBASE 中留出一个 PDMAP(4KB × 32)的大小，用来存储 fd（最多有 32 个 fd，每个 fd 占一个页面），确保这两部分的存储是不相重叠的

文件系统

总体架构

在整个文件系统的访问中，我们从用户进程到底层硬件，从上到下一次访问了这 6 个文件：

user / lib 目录：用户程序的库函数，允许用户程序使用统一的接口，抽象地操作文件系统中的文件，控制台，管道等虚拟文件

1. fd.c：实现文件描述符的相关操作
2. file.c：实现文件系统的用户接口
3. fsipc.c：实现用户进程与文件服务进程的交互

fs 目录：文件系统服务进程的代码：通过 IPC 通信与用户进程 user / lib / fsipc.c 内的通信函数进行交互

1. serv.c：进程的主干函数
2. fs.c：文件系统的基本功能函数
3. ide.c：通过系统调用和磁盘镜像交互

最后在最开始加载磁盘镜像的时候，初始化访问了一个 fsformat.c 文件，我们本次作业所有内容都在这 7 个文件之间进行交互

他们之间的逻辑关系如下：

```
用户态层（User Space）：
 ├── 文件描述符接口 fd.c       ← 用户使用 open/read/write/close
 |
 ├── 纯文件操作 file.c          ← 操作 Dev 接口的文件对象（对已开文件的操作结束于此）
 |     |
 |     ↓ 通过 fsipc.c（IPC）
 ├── 文件请求接口 fsipc.c      ← 将用户操作封装成 IPC 请求
 |     ↓
内核服务层（内核态）：
 ├── 文件服务进程 serv.c       ← 响应 IPC 请求，统一调度文件操作
 |     ↓
 ├── 文件系统逻辑 fs.c         ← inode、目录查找、文件块读写逻辑
 |     ↓
 └── 磁盘驱动 ide.c（不在五个中） ← 负责实际读写扇区和块缓存管理

```

各个文件职责分布如下：

文件名	模块层次	职责	向下调用
`fd.c`	用户空间（应用层）	提供统一的文件操作接口，如 `read/write/close/stat`。	调用 `dev_read/dev_write`，传递给 `file.c` 中的函数
`file.c`	用户空间（设备实现）	实现 `Dev` 抽象中的函数（file_read/file_write），调用 IPC 与文件服务通信。	调用 `fsipc.c`
`fsipc.c`	用户空间（IPC 封装）	发送 IPC 请求，向服务进程 `serv.c` 请求文件操作	发出 IPC 通信
`serv.c`	文件系统服务进程	接收 IPC 请求，分派到具体的 `fs.c` 逻辑处理	调用 `fs.c` 中的文件管理函数
`fs.c`	服务进程内部模块	管理文件描述表、文件元数据、块缓存等，负责文件与磁盘块的映射管理	调用 `ide.c`
`ide.c`	内核驱动	实现真正的磁盘扇区读写功能，直接访问设备地址	—

各个状态分区的解析：

概念名	定义	所在文件（作用）
①用户态定义与接口	用户调用的 `open()`，`read()`，`write()`，`close()` 接口函数	`fd.c`（统一接口，屏蔽下层细节）
②文件请求服务	用户请求通过 IPC 发送到文件服务进程	`fsipc.c`（封装请求/解析返回）
③文件进程服务	独立进程响应 IPC 请求，负责调用具体的文件操作	`serv.c`（类似于 RPC 的服务器）
④纯文件操作	调用 dev 抽象接口：`dev_read`，`dev_write`，`dev_close` 等函数	`file.c`（设备无关的抽象文件操作）
⑤文件操作	更深入的 inode 管理、目录解析、文件大小、分配文件块等	`fs.c`（负责 inode、路径解析、缓存等）
⑥磁盘块操作	实际对磁盘设备进行块或扇区的读取、写入，缓存页处理	`fs.c` 调用 `ide.c` 实现
⑦文件系统顶层	整体上协调所有文件系统功能模块，连接逻辑接口与硬件接口	由 `serv.c` 与 `fs.c` 协作体现

fd.c

fd.c 中维护了两类操作：

1. 和**文件描述符**有关的操作（fd 的创建，删除，分配，fdnum <-> data <-> fd 转换等），这部分操作是专门用来操作 fd 有关的
2. 和**文件描述符有关的**文件操作在**用户顶层**的接口（read，write等），这部分操作所面向的对象（文件，管道，控制台）对用户而言是不透明的，用户在调用相关方法的时候，并不知道操作的是

哪个设备

file.c

其实与之同级别的还有 console.c 和 pipe.c 这两个文件，分别负责对控制台和管道访问函数的具体实现，目前还涉及不到

file.c、console.c、ipie.c 这三个文件实现了 Dev 中的抽象函数，也就是根据访问设备的具体类型，设计与之相应的具体函数，比如对 write 而言，会分别设计针对文件，管道和控制台的三类 write 函数（lab5 中只涉及到**文件**操作）

也就是用户态下发一个统一的指令，在经过该层解析之后，就会变成一个针对于某个具体设备的指令

file.c 中维护了以下两类操作：

1. 对 fd.c 中那部分顶层函数的具体解析（file_* 类型的函数，比如 read -> file_read），这时候会把用户的宏观指令解析成针对于文件这一类设备的具体指令
2. 和**文件路径有关**的文件操作在用户顶层的接口（open，remove 等），会在 file.c 文件中先解析出 fd 来，再进行操作

注意：并不是所有文件操作都会下发给 fsipc.c 处理，比如 read 这类文件操作，不需要用底层的**块缓存**，可以在 file.c 本身完成

fsipc.c

文件的读写比较复杂，为了统一管理方便实现，我们把和文件有关的操作放到一个专门的**文件服务进程**中去实现

用户不能直接访问文件系统，必须通过 **IPC** 请求，发送给 serv.c 来完成文件操作，fsipc.c 可以认为是用户态到文件服务的桥梁，把用户态文件请求进行 IPC 格式化以后，交给文件服务进程处理，用户态通过一个**共用**的 fsipcbuf 页面，把请求以 req 结构体的形式传入 fsipc.c

serv.c

serv.c 即为**文件系统服务进程**，位于内核态，响应用户的 IPC 请求，本身相当于分发器，收到 IPC 请求以后把具体的文件操作**分发**给 fs.c 来实现

该文件在接收到用户请求以后，会为待操作的文件分配一个 Open 结构体，也就是文件打开窗口，只有通过这个窗口，内存在可以对文件进行一系列的操作

fs.c

fs.c 是**文集系统的逻辑核心**，负责实现具体的文件操作请求（read，write 等），为了避免频繁访问慢速磁盘，fs.c 会维护一个块缓存机制，加速访问速度，如果访问的文件块已经在缓存中，就直接返回，否则就调用 ide.c 去磁盘读扇区

这里所谓的**块缓存**，是一种 **“按磁盘块编号缓存到内存”** 的机制，每个磁盘块（通常大小为 512 字节或 4KB）对应缓存中的一个内存页，如果某个块已经缓存，就直接在内存中读/写，不需要触发 I/O

从具体实现上来看，我们使用 diskaddr 这个函数，把磁盘块线性映射到某个虚拟地址页，使得缓存在虚拟地址空间中被**懒加载**进来：

1. 调用 `diskaddr(blockno)` 获取块地址
2. 如果该地址没有被映射（即不在物理内存中），触发页异常（Page Fault）
3. 页异常处理函数会**调用 ide.c 从磁盘加载块到内存页**

fs.c 中的函数分为两类：

1. 以文件为单位操作的函数（基于块缓存函数的实现）
2. 以块缓存为单位操作的函数

ide.c

ide.c 是磁盘驱动，实现最底层的 I/O 操作，比如 IDE 接口的读写，所有的磁盘交互函数都是利用系统调用访问 KSEG1 段实现与外设的直接交互，并在最底层的函数中实现了以块、甚至扇区为单位的读写

该文件主要实现了以块为单位的磁盘的**读**和**写**操作，以及查询设备空闲状态的轮询函数

实例

以用户态对 read() 操作的访问为例：

```
用户进程：read(fd, buf, len)
      ↓
fd.c：找到对应 file 描述符
      ↓
fsipc.c：封装为 IPC_READ 请求，发送给 serv
      ↓
serv.c：接收请求，调用 file_read()
      ↓
file.c：根据文件类型调用 dev->read()
      ↓
fs.c：查找 inode、确定块号、调用 diskaddr(blockno)
      ↓
页异常触发 → 读磁盘块到内存页（由 ide.c 完成）
      ↓
返回数据 → 用户态完成读取
```

指导书设计

1. **外部存储设备驱动**：通常，我们需要按一定顺序读写**设备寄存器**，来实现对外部设备的操作，这里我们实现 IDE 磁盘的用户态驱动程序，该驱动程序将通过**系统调用**的方式陷入内核，对磁盘**镜像**进行读写操作
    - **设备寄存器**：硬件设备内部的一组特殊存储单元，用于控制设备行为或交换数据，CPU通过读写这些寄存器（通常是内存映射或端口I/O）来操作设备
    - **IDE**：一种磁盘接口标准，规定了一系列如寄存器映射，命令集，数据传输协议等规范
    - **磁盘镜像**：文件形式的虚拟磁盘（如`disk.img`），模拟真实磁盘的扇区结构，用来代替真正的物理外存硬件模拟文件系统
2. **文件系统的用户接口**：为用户提供接口和机制使得用户程序能够使用文件系统，这主要通过一个用户态的文件系统服务来实现，同时，我们将引入文件描述符等结构来抽象地表示一个进程所打开的文件，而不必关心文件实际的物理表示
3. **文件系统结构**：在本部分，我们会实现模拟磁盘的驱动程序以及磁盘上和操作系统中的文件系统结构，并实现文件系统操作的相关函数
    - fs 目录：文件系统服务程序的代码
    - user / lib 目录中： file.c、fd.c、fsipc.c 等文件存放文件系统的用户库，这部分代码会被一同链接到用户程序中，允许用户程序调用其中的函数来操作文件系统
    - 文件系统服务进程和其他用户进程之间使用 Lab4 中实现的 IPC机制进行通信（**注意**：**文件系统服务进程**实际上也是一个运行在用户态下的进程，而**其他用户进程**是指调用文件系统服务进程提供的 IPC 接口进行文件操作的程序，不包括操作系统提供的文件系统服务程序）

总体框架

1. **tools 目录**：存放构建时辅助工具的代码，我们将在其中实现 fsformat 工具，并借助它来创建磁盘镜像
2. **fs 目录**：存放文件系统服务进程的代码，我们在 fs.c 中实现文件系统的基本功能函数，在 ide.c 中通过系统调用与磁盘镜像进行交互，该进程的主干函数在 serv.c 中，通过 IPC 通信与用户进程 user / lib / fsipc.c 内的通信函数进行交互

3. **user/lib 目录**：存放用户程序的库函数，该目录下的 fsipc.c 实现了与文件系统服务进程的交互，file.c 实现了文件系统的用户接口，fd.c 中实现了文件描述符，允许用户程序使用统一的接口，抽象地操作磁盘文件系统中的文件，以及控制台和管道等虚拟的文件

整个文件系统主要分为以下三部分：

1. 将传统操作系统的文件系统移出内核，使用**用户态文件系统服务程序**和一系列**用户库**来实现，即使它们崩溃，也不会影响到整个内核的稳定，**其他用户进程**通过进程间通信(IPC) 来请求文件系统的相关服务，因此，在微内核中进程间通信 (IPC) 是一个十分重要的机制
2. 操作系统将一些内核数据暴露到用户空间，使得进程不需要切换到内核态就能访问，MOS 将进程页表映射到用户空间，此处文件系统服务进程访问自身进程页表即可判断磁盘缓存中是否存在对应块
3. 将传统操作系统的**设备驱动**移出内核，作为用户程序来实现，微内核体现在，内核在此过程中仅提供读写设备物理地址的系统调用

访问实例

注意：文件系统无论是内核态还是用户态的函数，如果要使用系统调用函数的话，必须都得调用 `syscall_*` 系列，而不能调用 `sys_*` 系列，因为后者没有对应的头文件，无法在 `syscall_all.c` 文件之外被调用

remove 系列

我们以对文件的 remove 操作为例，串一下整个文件系统的访问流程，删除文件不需要提前打开文件，因此是根据路径 path 触发（可以删除 fd 中没有标记的那些文件）

remove

函数功能：没什么好解析的，直接调用下一级的 fsipc 函数，进行 IPC 通信

```
int remove(const char *path) {
        return fsipc_remove(path);
}
```

fsipc_remove

函数功能：解析全局数组 fsipcbuf -> req，初始化操作字段 req->path，把要删除的路径向下传递到 serv 分发函数

```
int fsipc_remove(const char *path) {
// 路径合法检查 : 不能为空路径，不能超过最大长度限制
        if (path[0] == '\0' || strlen(path) >= MAXPATHLEN) {
                return -E_BAD_PATH;
        }

// 类型强转 : 告知系统，要处理的是 remove 请求，请把 fsipcbuf 当成存放 remove 请求字段的页面来解析
        struct Fsreq_remove *req = (struct Fsreq_remove *)fsipcbuf;

// 把要删除的文件路径存入 fsipcbuf 的 path 中
        strcpy((char *)req->req_path, path);

// 调用 fsipc 函数，把文件删除操作传递给文件服务系统，这里不需要填写 fd 结构体，因为文件都删没了，后两个参数都是 0
        return fsipc(FSREQ_REMOVE, req, 0, 0);
}
```

serve_remove

函数功能：只起中转操作的作用，调用 file_remove 执行真正的删除操作，再把删除操作的结果通过 ipc_send 返回给用户进程

```
void serve_remove(u_int envid, struct Fsreq_remove *rq) {
        int r = file_remove(rq->req_path);
        ipc_send(envid, r, 0, 0);
}
```

file_remove

函数功能：封装好的文件操作，底层会调用块缓存操作函数，执行真正的文件删除功能

```
int file_remove(char *path) {
        int r;
        struct File *f;
// 找到 path 所对应那个的文件的文件控制块
        if ((r = walk_path(path, 0, &f, 0)) < 0) {
                return r;
        }

// 调用 file_truncate，把文件大小减到 0，相当于内容的删除
        file_truncate(f, 0);

// 清空文件名数组，相当于文件名删除
        f->f_name[0] = '\0';

// 调用 file_flush，把删除后的空文件写回到磁盘中，如果文件有上级目录，上级目录也要写回一遍
        file_flush(f);
        if (f->f_dir) {
                file_flush(f->f_dir);
        }

        return 0;
}
```

用户对文件的操作大致可以分成两类，一类是对**未打开文件**的操作，一类是对**已打开文件**的操作

1. 对未打开文件的操作：open，remove，ftruncate，sync
2. 对已打开文件的操作：sta，fstat，seek，write，readn，read，dup，close，close_all，read_map

其中对**未打开文件**的操作，因为他们之前没有打开过，也就是内核的全局打开文件表中没有他们的登记，用户态的文件描述符表中也没有他们的登记，因此我们需要通过**路径**（path）对他们进行访问，并且在访问后把相关信息存入用户态中

其中对**已打开文件**的操作，因为他们曾经被打开过，内核的全局打开文件表和用户态的文件描述符表中都有他们的登记，我们**没必要**再回内核态去物理内存甚至是底层磁盘中访问，直接访问文件描述符结构体即可得到相关信息，因此这些函数往往在 file.c 文件中，也就是用户态下就可以直接实现相应的功能

还有一种区分这两种操作的方法，即观察 file_* 函数所在的位置，如果操作的是未打开文件，比如 open，那么 file_open 函数会实现在 fs.c 内核态，而如果操作的是已打开文件，比如 read，那么 file_read 函数会实现在 file.c 用户态，这也是为什么很多参考资料会把 file.c 这个文件的调用顺序排在 serv.c 之后

open 系列

open（在用户态打开文件）可以称得上是整个文件系统最核心的一个文件操作，对于了解文件系统层次和相关函数调用至关重要，下面进行简单解析

file.c / open

```
// 函数功能 : 打开文件，参考文件路径为 path，打开方式为 mode
int open(const char *path, int mode) {
        int r;

// 使用 fd_alloc 为新打开的文件分配一个 fd 结构体
        struct Fd *fd;
        r = fd_alloc(&fd);
        if (r) {
                return r;
        }
// 通过调用 fsipc_open 填好 fd 中的相应内容(设备类型，操作位置偏移量，打开模式)
        r = fsipc_open(path, mode, fd);
        if (r) {
                return r;
        }
/* 设置用户态打开文件的四个参数
    1. 虚拟地址 va 用来存储打开文件的文件内容，fd2data(fd)
    2. ffd 存储用户态打开文件结构体，(struct Filefd *)fd
    3. size 表示打开文件的大小，ffd->file.f_size
    4. fileid 表示打开文件的编号，ffd->f_fileid
*/
        char *va;
        struct Filefd *ffd;
        u_int size, fileid;
        va = fd2data(fd);
        ffd = (struct Filefd *)fd;
        size = ffd->f_file.f_size;
        fileid = ffd->f_fileid;
// 通过调用 fsipc_map 映射文件的具体内容，这个函数后面会具体解析，只需要知道它的功能就是把文件内容映射到 va 对应的虚拟地址中即可
        for (int i = 0; i < size; i += PTMAP) {
                r = fsipc_map(fileid, i, va + i);
                if (r) {
                        return r;
                }
        }
// 把得到的文件描述符编号 fdnum 返回
        return fd2num(fd);
}
```

fd 指针怎么理解：可以认为 fd 是指向某个页面的地址，这个页面中用来存储打开文件的有关信息（struct Filefd 的字段），在使用 fd_alloc 的时候，系统就会开辟一个内存页，而 fd 就是这个页面的地址，同时这个页面也是用户进程和内核进程之间 IPC 通信的共享页面，fd 结构体具体字段的填写就是通过共享页面，现在内核态把当前页面填写完成，再传回用户态所实现的

我们需要了解，open 函数怎么能够把**内核态**的文件信息传递回用户态，其实是分了 5 个步骤：现在假设我们已经把打开文件的请求传到了 serv.c 中（传入的过程是很好理解的，重点是信息传出的过程）

1. 在内核态为待打开的文件分配一个 Open 结构体，添加一条文件在内存中被打开的**全局打开**记录
2. 调用 fs.c 中的内核函数，分配指定的文件控制块 file，同时调用磁盘块操作函数，把文件在对应磁盘中的内容加载到磁盘缓冲区中
3. 使用 file 结构体的字段，填写 open 结构体的字段，这样我们在内核态就拿到了打开文件的相关信息
4. 在用户态调用 open 函数之初，操作系统就会为该文件分配一个页面，也就是 fd 指针所指向的地址（这里的意思是 fd 这个结构体实际上是以一个页面形式存在的），这个页面还同时作为文件服务进程和用户进程之间的共享页面
5. 用户进程和文件服务进程通过 IPC 进行交互，首先用户进程把 fd 这个页面地址通过 IPC 传入内核，在 srv.c 中完成有关 Open 结构体字段对 Fd结构体字段的赋值，内核把填好的 fd 页面再回传给用户态，此时 fd 结构体中就已经存好了文件的相关信息
6. 每个 fd 页面都会**线性映射**一个 data 页面，在初始化 fd 的时候，通过 fd2data 随之也初始化一段虚拟地址空间 va，这段虚拟地址空间专门用来作为文件缓冲区，也就是存储 fd 所表示的那个文件的内容
7. 在调用 fsipc_open 函数的时候，只完成 va 空间的初始化，而文件内容真正填入缓冲区实际上是通过 fsipc_map 实现的，这里我们巧妙的使用了 fsipc_map 入参的含义，它可以把 fileid 对应的文件中偏移量为 offset 的页面写入 va 对应的虚拟地址中，这里通过一个循环，就能够把文件的页面**线性保序**映射到虚拟地址中

fsipc.c / fsipc_open

```c
// 函数功能 : 把打开文件的请求传递给 serv.c，传入参数为文件的路径 path，打开方式 omode，把为打开文件分配的 Fd 结构体存入 fd 指针中
int fsipc_open(const char *path, u_int omode, struct Fd *fd) {
        u_int perm;
        struct Fsreq_open *req;
// 告诉操作系统请按照 Fsreq_open 的形式解读 fsipcbuf，把他解析成文件打开请求
        req = (struct Fsreq_open *)fsipcbuf;

// 非法路径
        if (strlen(path) >= MAXPATHLEN) {
                return -E_BAD_PATH;
        }

// req 中存入待传入内核的参数 : 文件路径和打开方式
        strcpy((char *)req->req_path, path);
        req->req_omode = omode;
// 核心 : 调用 fsipc 向内核态 serv_open 发送操作请求，同时把操作结果返回用户态 open
        return fsipc(FSREQ_OPEN, req, fd, &perm);
}
```

fsipc / fsipc.c

和整个文件共用同一个名字的函数，含金量就不用说了吧，这个函数负责所有来自用户态的文件操作请求向内核态的转发

```c
// 函数功能 : 把用户态的文件操作请求传递给内核态，把内核态的处理结果返回用户态，入参依次为请求类型、请求结构体、待填写的文件描述符所在页面，权限位标志
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm) {
        u_int whom;
// 我们的操作系统硬性指定，文件服务系统进程就是 envs 表中的 1 号进程
        ipc_send(envs[1].env_id, type, fsreq, PTE_D);
        return ipc_recv(&whom, dstva, perm);
}
```

serv.c / serve_open

进入 serv.c 文件以后，文件操作就正式进入内核态，serv_open 相当于是内核的第一道门户，接收来自用户态的 IPC 请求，但自己不处理，而是调用内核态下层的函数处理，处理结果仍然以 IPC 形式返回用户态

```c
// 函数功能 : 根据发出请求的进程 id 和请求结构体 rq 中的参数，调用 fs.c 中的底层函数打开目标文件，该函数不需要返回值，因为打开文件的 fd 会通过 IPC 以共享页面的形式传回
void serve_open(u_int envid, struct Fsreq_open *rq) {
        struct File *f;
        struct Filefd *ff;
        int r;
        struct Open *o;

// 调用 open_alloc，在内核态为待打开的文件申请一个打开文件结构体 o，用来在内核态存储打开文件的信息
        if ((r = open_alloc(&o)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

// 检查权限问题 : 如果要求在打开时同时创建文件，且创建文件失败(不是因为文件已存在才失败)，直接报错返回
        if ((rq->req_omode & O_CREAT) && (r = file_create(rq->req_path, &f)) < 0 &&
            r != -E_FILE_EXISTS) {
                ipc_send(envid, r, 0, 0);
                return;
        }

// 调用 fs.c 中的 file_open 执行文件打开，同时填写 File 结构体 f 的内容，这个内容会继续被填入 Open 结构体，最后填入 ff 结构体被共享返回
        if ((r = file_open(rq->req_path, &f)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

// 在 Open 结构体中记录文件控制块 f
        o->o_file = f;

// 检查权限问题 : 是否需要创建空文件，也就是调用 set_file_size 把文件大小清空
        if (rq->req_omode & O_TRUNC) {
                if ((r = file_set_size(f, 0)) < 0) {
                        ipc_send(envid, r, 0, 0);
                }
        }

```

```
/* 最关键的一步，根据 f 填写 o，根据 o 填写 ff，把 ff 作为共享页面通过 IPC 返回，这里要填写的字段很多，返回的权限位是已修改+共享页面
*/
        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, o->o_ff, PTE_D | PTE_LIBRARY);
}
```

通过观察 serv_open 函数，我们可以得知，真正来自于 fs.c 底层操作为文件打开的贡献主要是两个，而这两个功能则需要调用很多底层函数来逐一实现：

1. 获得了目标文件的文件控制块，并将其赋值给 open 结构体的 o_ff 字段
2. 如果文件在磁盘中没有映射，则建立文件和磁盘之间的映射

fs.c / file_open

进入 fs.c 文件，所调用的函数就是与具体**文件**操作有关的函数（文件操作还会向下调用磁盘块操作实现）

```
// 函数功能 : 打开路径 path 所指向的文件，对应的文件控制块存入 file 指针中
int file_open(char *path, struct File **file) {
    return walk_path(path, 0, file, 0);
}
```

fs.c / walk_path

```
// 函数功能 : 根据路径打开文件，入参体现了打开成功与否的两种情况，如果打开成功，就用 pfile 存打开文件的文件控制块，用 pdir 存打开文件上级目录的文件控制块；打开失败就用 lastelem 存路径最后一段的名称
int walk_path(char *path, struct File **pdir, struct File **pfile, char *lastelem) {
        char *p;
        char name[MAXNAMELEN];
        struct File *dir, *file;
        int r;

// 从根目录开始解析路径，首先 file 指向根目录的文件控制块(可以通过 super->s_root 获得)，同时 path 路径跳过最开头根目录代表的 / (如果要查找相对目录，则 file 直接指向相对根目录，path 不用动就行)
        path = skip_slash(path);
        file = &super->s_root;

// 初始化目标文件指针，上级目录指针和文件名都指向空(这里 dir 对应 pdir，file 对应 pfile，只不过前者是用来遍历的，后者是存返回答案的)
        dir = 0;
        name[0] = 0;
        if (pdir) { *pdir = 0; }
        *pfile = 0;

// 一段一段解析 path，按照 / 进行划分
        while (*path != '\0') {
```

```
            // 每解析到一段，上一段就变成当前当的上级目录(pre_file->tmp_dir)，同时记录一下当前剩余的路径(用来构造 lastelem)
                dir = file;
                p = path;
            // 路径跳过 / ，如果路径长度超过范围，直接报错返回
                while (*path != '/' && *path != '\0') { path++; }
                if (path - p >= MAXNAMELEN) { return -E_BAD_PATH; }
            // 从 p 到 path - p 的一段记录到 name 数组中，同时设置 name 数组的结尾 \0
                memcpy(name, p, path - p);
                name[path - p] = '\0';
                path = skip_slash(path);
            // 如果发现上级目录不是目录类型，直接报错返回(文件必须存在于一个合法目录下)
                if (dir->f_type != FTYPE_DIR) {
                        return -E_NOT_FOUND;
                }
            // 调用 dir_lookup，在上级目录中查找名为 name 的文件
                if ((r = dir_lookup(dir, name, &file)) < 0) {
                        if (r == -E_NOT_FOUND && *path == '\0') {
                                if (pdir) { *pdir = dir; }
                                if (lastelem) { strcpy(lastelem, name); }
                                *pfile = 0;
                        }
                        return r;
                }
        }
// 记录上级目录 pdir 为 dir，目标文件 pfile 为 file
        if (pdir) { *pdir = dir; }
        *pfile = file;
        return 0;
}
```

注意这里的路径查找是一个依次查找的过程，比如 A / B / C / target，有可能在 A 目录下查找 B 就找不到，这时候后面就不用再看了，lastelem 存的就是 A，也就是给出目录和实际目录重合的最后一段，正是因为有这种情况，我们才要在 path 解析的每一段都是用 dir_lookup 进行查找

fs.c / dir_lookup

从宏观上来讲，到 dir_lookup 为止，和文件打开的过程就结束了，只是该函数又继续调用了一些底层的磁盘映射函数，因此后面又进行了简单的子函数解析

```
// 函数功能 : 在 dir 目录下，查找名为 name 的文件或目录，如果找到，目标文件控制块存入 file 中
int dir_lookup(struct File *dir, char *name, struct File **file) {
// 首先要知道，当前目录文件 dir 下有多少磁盘块
        u_int nblock;
        nblock = dir->f_size / BLOCK_SIZE;
      // 接着遍历这些所有的磁盘块，对于每个控制块，再遍历里面的文件控制块
        for (int i = 0; i < nblock; i++) {
                void *blk;
          // 调用 file_get_block 函数，获取 dir 目录文件下第 i 个文件块的内容，blk 指向它映射内存的虚拟地址(具体实现是先找到第 i 个文件块对应的磁盘块，再找到这个磁盘块在内存中的映射，最终返回的是映射后的地址)
                try(file_get_block(dir, i, &blk));
          // 因为 dir 是目录文件，那么它所包含的每一个块都应该是目录块，也就是由目录项(文件控制块)所组成的磁盘块，下面我们要遍历每个目录项(文件控制块)，也就是以 File 大小为步长，遍历刚刚找到的 blk 这个目录块
                struct File *files = (struct File *)blk;
```

```
                // 这里又用了指针步长的相关知识，f 每次会跳过一个 struct File 的大小
                for (struct File *f = files; f < files + FILE2BLK; ++f) {
                        // 比较文件名称是否相同，如果是，则记录当前文件控制块就是目标文件的文件控制块，同时记录它的上级目录是入参 dir 即可
                        if (strcmp(name, f->f_name) == 0) {
                                *file = f;
                                f->f_dir = dir;
                                return 0;
                        }
                }
        }
        return -E_NOT_FOUND;
}
```

文件块与磁盘块的遍历

1. 对于某个目录文件，先遍历这个目录文件中的所有目录块，再遍历每个目录块下的所有目录项
2. 对于某个文件控制块，遍历它所能索引到的每个磁盘块号（包括直接索引和间接索引）

```
// 遍历某个文件控制块 dirf 之下的所有磁盘块号(直接和间接索引)
sruct File *f;
void *blk;
int nblock = f->f_size / BLOCK_SIZE;
for (int i = 0; i < nblk; i++) {
  int bno; // 磁盘块号
  if (i < NDIRECT) {
    bno = dirf->f_direct[i];
  } else {
    bno = (disk[dirf->f_indirect].data)[i];
  }
  // do sth to bno
}
```

在得到磁盘块以后，如果想访问磁盘块的内容，有以下两种方式：

1. 直接借助 disk [bno].data 段得到磁盘块内容
2. 调用 read_block(bno, &blk) 把磁盘块 bno 对应的内容存入虚拟地址 blk 中

```
// 遍历某个文件控制块 dirf 之下所有的磁盘块(获得它在内存缓冲区中的地址)，进而遍历每个磁盘块中的所有文件控制块
sruct File *f;
void *blk;
int nblock = f->f_size / BLOCK_SIZE;
for (i = 0; i < nblock; i++) {
    try(file_get_block(dir, i, &blk));
    f = (struct File *)blk;
    for (j = 0; j < FILE2BLK; j++) {
      // f[j]
        }
    }
}
```

这里需要把 blk 解释成为 File* 类型的数组，这样每次 j++ 所能跳过的就是一个文件控制块的步长

```
// 遍历一个目录下的所有目录块，进而遍历每个目录块中的所有文件控制块
for (int i = 0; i < nblock; i++) {
  void *blk;
  try(file_get_block(dir, i, &blk));
  struct File *files = (struct File *)blk;
  for (struct File *f = files; f < files + FILE2BLK; ++f) {
    // f
    }
  }
}
```

这种实现方式和上面的实现方式是等价的，只是在遍历每个磁盘块中的文件控制块的具体实现上稍有区别

```
// 在某个文件控制块中查找 filebno 这个文件块所对应的磁盘块号 / 磁盘块号的地址
if (filebno < NDIRECT) {
  ptr = &f->f_direct[filebno];
} else if (filebno < NINDIRECT) {
  read_block(f->f_indirect, (void **)&blk, 0)
  ptr = blk + filebno;
  // if need bno, blk[filebno]
}
*ppdiskbno = ptr;
```

需要注意的是，一个 File 文件控制块所能所引到的磁盘块号的数量和一个目录文件下的目录块的数量往往是不同的，后者可以用 f_size / BLOCK_SIZE 所得到，而前者只能是根给出的文件块编号 filebno 来卡他所在的磁盘块号

前三种实现方式和最后一种实现方式，对磁盘块的解释不同，前者需要把遍历到的磁盘块解释成 File* 类型的数组，每个元素都是一个文件控制块；最后一种需要把间接块 f_indirect 解释成 u_int 类型的数组，每个元素都是一个磁盘块整数

fs.c / file_get_block

```
// 函数功能 : 查找文件控制块 f 所代表的文件的第 i 个文件块在内存中数据映射的地址，既然是查找，那么就要求没有磁盘块的就映射磁盘块，没有缓存的就映射缓存，这一步也是[首次访问加载文件内容到内存]的过程
int file_get_block(struct File *f, u_int filebno, void **blk) {
        int r;
        u_int diskbno;
        u_int isnew;

        // 先找到文件块对应的磁盘块
        if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) {
                return r;
        }

        // 再找到磁盘块所对应的缓存所映射的虚拟地址
        if ((r = read_block(diskbno, blk, &isnew)) < 0) {
                return r;
        }
        return 0;
}

```

注意区分**磁盘缓冲区**和**文件内容映射区**，也就是说文件内容是有两次映射的，在打开文件的时候，第一次是处理 fsipc_open 调用 file_get_block 的时候，此时文件会先被映射到磁盘块，然后映射到磁盘缓冲区，也就是 diskaddr 函数所完成的线性映射；第二次是处理 fsipc_map 调用的时候，再次调用 file_get_block，此时会把磁盘缓冲区中的文件内容映射到由 fd 线性索引分配得到的 data 段，作为文件内容在用户态下的存储

1. 磁盘块 -> 磁盘缓冲区：内核态可以访问文件
2. 磁盘缓冲区 -> 文件缓冲区：用户态可以访问文件

fs.c / file_map_block

```
// 函数功能 : 查找文件控制块 f 对应的文件，第 filebno 个文件块，在磁盘中对应的磁盘块号 diskbno，以及如果不存在映射且 alloc 为 1 时，需要建立映射
int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc) {
        int r;
        uint32_t *ptr;

// 调用 file_block_walk 首先查找 filebno 是否已经存在某个磁盘块的映射
        if ((r = file_block_walk(f, filebno, &ptr, alloc)) < 0) {
                return r;
        }

// 如果尚且不存在磁盘块映射，且 alloc 为 1，则调用 alloc_block 分配一个新的磁盘块
        if (*ptr == 0) {
                if (alloc == 0) {
                        return -E_NOT_FOUND;
                }

                if ((r = alloc_block()) < 0) {
                        return r;
                }
                *ptr = r;
        }

// 最后建立新的映射即可，注意这里要求的是值的改变，也就是 diskbno 这个指针所指向的地址中的内容，改成 ptr 这个指针所指向的地地址中的内容，如果写成 diskbno = ptr 则变成了让 diskbno 指向 ptr 所指向的地址，相当于修改了指针的指向，别忘了 diskbno 是个指针形式的形参，在它同级别进行修改，调用者是得不到反馈的
        *diskbno = *ptr;
        return 0;
}
```

fs.c / read_block

```
// 函数功能 : 获取磁盘块号为 blockno 的磁盘在对应的内存虚拟地址，把对应的磁盘块内容[读入]到内存中，如果是首次访问，isnew 需要设为 1，要想访问某个磁盘块中的 File 项，必须把它拿到内存中操作，不能在磁盘中直接操作之
int read_block(u_int blockno, void **blk, u_int *isnew) {
// 检验磁盘块号合法性 : 超级块存在，且该块号不超过最大磁盘数；位示图存在且该块号对应的磁盘块不空，此时才能读取
        if (super && blockno >= super->s_nblocks) {
                user_panic("reading non-existent block %08x\n", blockno);
        }
        if (bitmap && block_is_free(blockno)) {
                user_panic("reading free block %08x\n", blockno);
```

```
        }

// 调用 diskaddr 获取磁盘块号对应的虚拟地址
        void *va = disk_addr(blockno);

// 调用 bolck_is_mapped，本质上是判断 va 是否通过页表和某个物理地址建立映射，判断当前磁盘块 blockno 之前是否被访问过
        if (block_is_mapped(blockno)) {
                if (isnew) {
                        *isnew = 0;
                }
        } else {
                if (isnew) {
                        *isnew = 1;
                }
          // 磁盘块内容的拷贝，注意 ide_read 的第二个参数是物理地址，不要画蛇添足加 KSEG1
                try(syscall_mem_alloc(0, va, PTE_D));
                ide_read(0, blockno * SECT2BLK, va, SECT2BLK);
        }

// 记录我们找到的虚拟地址 va
        if (blk) {
                *blk = va;
        }
        return 0;
}
```

fs.c / file_block_walk

```
// 查找文件控制块 f 对应的文件，编号为 filebno 的文件块对应的磁盘块号的[地址]，或者可以认为是指向这个磁盘块号的指针，如果不存在映射且 alloc 为 1 时要分配一个磁盘块
// 这里看似很奇怪，返回的是磁盘块号的地址，而不是磁盘块号本身，因此要用二级指针来存储
int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int alloc) {
        int r;
        uint32_t *ptr;
// 函数体内部操作的都是一级指针
        uint32_t *blk;

        if (filebno < NDIRECT) {
        // 直接映射的情况，如果是直接映射，直接取地址返回即可
                ptr = &f->f_direct[filebno];
        } else if (filebno < NINDIRECT) {
          // 间接映射的情况，如果目前还没有间接映射指针，根据 alloc 决定是否分配，同时还要检查分配是否成功
                if (f->f_indirect == 0) {
                        if (alloc == 0) {
                                return -E_NOT_FOUND;
                        }

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

```

```
// 调用 read_block，把间接块读入内存，blk 是这个间接块在进程空间中的虚拟地址，此时我们把间接块(本质上也是一个磁盘块)解释成为一个存储了 1024 个磁盘块号的数组，把他当数组进行索引访问
                if ((r = read_block(f->f_indirect, (void **)&blk, 0)) < 0) {
                        return r;
                }
          // ptr 指向间接块中第 filebno 个元素，也就是文件的逻辑块号在间接块中对应的磁盘块号，这里涉及指针步长，实际上加的是 filebno × BY2BLK 这么大的空间，等价于 ptr = &blk[filebno];
                ptr = blk + filebno;
        } else {
                return -E_INVAL;
        }
    *ppdiskbno = ptr;
        return 0;
}
```

我们回看一下在 file_map_block 函数中调用 file_block_walk 的过程，因为前者的给后者的参数是一个一一级指针的取址，因此后者的形参必须设计成二级指针，这么设计的本质原因是，我们想通过一个一级指针记录磁盘块号，但是这个磁盘块号在求解的时候又经过了一次函数调用，因此一级指针变成了二级指针

close 系列

fd.c / close

```
int close(int fdnum) {
        int r;
        struct Dev *dev = NULL;
        struct Fd *fd;

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

        r = (*dev->dev_close)(fd);
        fd_close(fd);
        return r;
}
```

file.c / file_close

```
int file_close(struct Fd *fd) {
        int r;
        struct Filefd *ffd;
        void *va;
        u_int size, fileid;
        u_int i;

        ffd = (struct Filefd *)fd;
        fileid = ffd->f_fileid;
        size = ffd->f_file.f_size;

        va = fd2data(fd);

```

```
        for (i = 0; i < size; i += PTMAP) {
                if ((r = fsipc_dirty(fileid, i)) < 0) {
                        debugf("cannot mark pages as dirty\n");
                        return r;
                }
        }

        if ((r = fsipc_close(fileid)) < 0) {
                debugf("cannot close the file\n");
                return r;
        }

        if (size == 0) {
                return 0;
        }
// 关闭文件，别忘了把文件在内存中的缓存也清楚，取消掉对应虚拟地址和物理页面的映射即可
        for (i = 0; i < size; i += PTMAP) {
                if ((r = syscall_mem_unmap(0, (void *)(va + i))) < 0) {
                        debugf("cannont unmap the file\n");
                        return r;
                }
        }
        return 0;
}
```

fsipc.c / fsipc_dirty

```
int fsipc_dirty(u_int fileid, u_int offset) {
        struct Fsreq_dirty *req;

        req = (struct Fsreq_dirty *)fsipcbuf;
        req->req_fileid = fileid;
        req->req_offset = offset;
        return fsipc(FSREQ_DIRTY, req, 0, 0);
}
```

fsipc.c / fsipc_close

```
int fsipc_close(u_int fileid) {
        struct Fsreq_close *req;

        req = (struct Fsreq_close *)fsipcbuf;
        req->req_fileid = fileid;
        return fsipc(FSREQ_CLOSE, req, 0, 0);
}
```

serv.c / serv_dirty

```
void serve_dirty(u_int envid, struct Fsreq_dirty *rq) {
        struct Open *pOpen;
        int r;

        if ((r = open_lookup(envid, rq->req_fileid, &pOpen)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
```

```
        }

        if ((r = file_dirty(pOpen->o_file, rq->req_offset)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

        ipc_send(envid, 0, 0, 0);
}
```

serv.c / serv_close

```
void serve_close(u_int envid, struct Fsreq_close *rq) {
        struct Open *pOpen;

        int r;

        if ((r = open_lookup(envid, rq->req_fileid, &pOpen)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

        file_close(pOpen->o_file);
        ipc_send(envid, 0, 0, 0);
}
```

fs.c / file_dirty

```
int file_dirty(struct File *f, u_int offset) {
        int r;
        u_int diskbno;

        if ((r = file_map_block(f, offset / BLOCK_SIZE, &diskbno, 0)) < 0) {
                return r;
        }

        return dirty_block(diskbno);
}
```

fs.c / file_close

```
void file_close(struct File *f) {
// 先把缓冲的内容刷新回磁盘中
        file_flush(f);
// 判断要关闭的文件是否属于某个目录
        if (f->f_dir) {
          // 遍历这个目录下的文件块，找到这个文件对应的文件控制块
                u_int nblock = f->f_dir->f_size / BLOCK_SIZE;
                for (int i = 0; i < nblock; i++) {
                        u_int diskbno;
                        struct File *files;
                  // 找到第 i 个文件块在内存中具体的内容(文件控制块->具体的块内容)，把他解释称 File 类型结构体数组(里面存的全都是 File)
                        if (file_map_block(f->f_dir, i, &diskbno, 0) < 0) {
```

```
                                debugf("file_close: file_map_block
failed\n");
                                break;
                        }
                        if (read_block(diskbno, (void **)&files, 0) < 0) {
                                debugf("file_close: read_block failed\n");
                                break;
                        }
                    // 比较要修改的那个文件控制块是不是在这个文件块的范围内，如果是，那么
这个文件块就相应变成了[脏块]，需要设置标志位
                        if (files <= f && f < files + FILE2BLK) {
                                dirty_block(diskbno);
                                break;
                        }
                }
// 刷新上级目录，确保它的修改都被刷新回磁盘
                file_flush(f->f_dir);
        }
}
```

这里需要注意一点误区：通过上级目录不能**直接找到**下级文件，找到的只能是下级文件的**文件控制块**，而要想找到下级文件的具体内容，需要根据适才索引到的 File 接着调用 file_get_block（内部又封装了 read_block 和 file_map_block 这两个函数），才能找到文件在磁盘缓冲区中映射的内容

而 file_get_block 也只是负责找到某个文件块的内容，如果想找到一个文件控制块所对应的文件的全部内容的话，就要遍历它所有的文件块号，也就是要分别遍历直接索引和间接索引两部分，对于每个文件块号都访问衣服 file_get_block

fs.c / dirty_block

```
// 标记编号为 blockno 的磁盘块，对应在内存中物理页被修改过
int dirty_block(u_int blockno) {
        void *va = disk_addr(blockno);

        if (!va_is_mapped(va)) {
                return -E_NOT_FOUND;
        }

        if (va_is_dirty(va)) {
                return 0;
        }
        // 自映射，设置 PTE_D 标志位即可
        return syscall_mem_map(0, va, 0, va, PTE_D | PTE_DIRTY);
}
```

map 系列

read_map

```
// 函数功能 : 读取 fdnum 对应的文件，偏移为 offset 位置处的页面，把读取的结果存入 blk
对应的虚拟地址空间中(此时确保文件在内存中已经有打开记录)
int read_map(int fdnum, u_int offset, void **blk) {
        int r;
        void *va;
        struct Fd *fd;

```

```
        if ((r = fd_lookup(fdnum, &fd)) < 0) {
                return r;
        }

        if (fd->fd_dev_id != devfile.dev_id) {
                return -E_INVAL;
        }
// 这里文件内容是直接读出来的，没有再进入内核
        va = fd2data(fd) + offset;

        if (offset >= MAXFILESIZE) {
                return -E_NO_DISK;
        }

        if (!(vpd[PDX(va)] & PTE_V) || !(vpt[VPN(va)] & PTE_V)) {
                return -E_NO_DISK;
        }

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

fsicp_map

```
// 函数功能 : 把文件编号为 fileid 的文件偏移 offset 处的页面，映射到 dstva 所对应的虚拟地址空间中(这个函数在 open 建立文件和缓冲区映射的时候得到了巧妙应用)
int fsipc_map(u_int fileid, u_int offset, void *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_D | PTE_LIBRARY)) != (PTE_V)) {
                user_panic("fsipc_map: unexpected permissions %08x for dstva %08x", perm, dstva);
        }

        return 0;
}
```

serve_map

```
// 函数功能 : 完成 envid 对应进程，有关文件映射的请求 rq
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 / BLOCK_SIZE;

        if ((r = file_get_block(pOpen->o_file, filebno, &blk)) < 0) {
                ipc_send(envid, r, 0, 0);
                return;
        }

        ipc_send(envid, 0, blk, PTE_D | PTE_LIBRARY);
}
```

值得注意但是，read_map 这个函数才是在用户态读取一个文件页到指定虚拟地址的函数（直接操作 fd 对应的内存），而 fsipc_map 和 serve_map 则是用来服务于 open 函数把文件内容映射到缓存区用的，切忌望文生义

read 系列

read

函数功能：根据文件描述符，找到对应的设备类型，前往该设备对应的具体操作函数执行，把宏观文件操作具体化

```
int read(int fdnum, void *buf, u_int n) {
        int r;
        struct Dev *dev;
        struct Fd *fd;
// 根据 fdnum 查找到文件描述符 fd，进而根据 fd 的设备 id 字段找到对应的设备
        if ((r = fd_lookup(fdnum, &fd)) < 0) {
                return r;
        }
        if ((r = dev_lookup(fd->fd_dev_id, &dev)) < 0) {
                return r;
        }

// 检查操作权限是否合法
        if ((fd->fd_omode & O_ACCMODE) == O_WRONLY) {
                return -E_INVAL;
        }

// 根据找到的设备，调用该设备对应 read 操作的函数指针，进入 file.c 中进行 file_read 函数
        r = dev->dev_read(fd, buf, n, fd->fd_offset);

// 如果操作成功，r 表示读取的字节数，文件操作的偏移量增加 r，表示下次该读取的位置
        if (r > 0) {
                fd->fd_offset += r;
        }
        return r;
}
```

file_read

函数功能：直接完成对文件内容的读取，不会再向下一级传递，也就是说对于 read 这个操作，不会传递到底层的**块缓存**层面完成

```
static int file_read(struct Fd *fd, void *buf, u_int n, u_int offset) {
        u_int size;
        struct Filefd *f;
        f = (struct Filefd *)fd;

        size = f->f_file.f_size;

        if (offset > size) {
                return 0;
        }

        if (offset + n > size) {
                n = size - offset;
        }

        memcpy(buf, (char *)fd2data(fd) + offset, n);
        return n;
}
```

结构体汇总

Block

```
struct Block {
  uint8_t data[BY2BLK]; // 一个磁盘块大小为 512B
  uint32_t type;// 磁盘块类型(目录块 or 数据块)
} disk[NBLOCK];
```

Open

```
struct Open {
  struct File *o_file; // 当前打开文件的文件控制块
  u_int o_fileid;      // 文件系统分配的唯一文件 ID，表示当前文件在 opentab 中的下标
  int o_mode;          // 文件的打开模式
  struct Filefd *o_ff; // 用户态打开文件结构体
};

struct Open opentag[MAXOPEN];// 管理所有 Open 标识符的数组
```

File

```c
struct File {
    char f_name[MAXNAMELEN]; // 文件名
    uint32_t f_size; // 文件大小
    uint32_t f_type; // 文件类型
    uint32_t f_direct[NDIRECT];  // 直接索引
    uint32_t f_indirect; // 间接索引
    struct File *f_dir;  // 父目录
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));

#define BY2FILE 256
#define MAXNAMELEN 128
```

Fd

```c
struct Fd {
  u_int fd_dev_id;  // 此设备对应的 Dev 类型
  u_int fd_offset;  // 目前的文件指针距离起始的偏移量
  u_int fd_omode; // 设备的打开模式
};
```

Filefd

```c
struct Filefd {
  struct Fd f_fd; // 文件描述符的通用部分(设备、偏移、打开模式)
  u_int f_fileid; // 文件 ID(这个字段很重要，它代表这个文件在内核中 opentab 数组中的下标)，通过这个 f_fileid 可以访问到 Open 结构体的对应信息
  struct File f_file; // 具体文件元数据(文件名，类型，大小，直接块，间接块等)
};
```

思考题

5.1

> **Thinking 5.1** 如果通过 kseg0 读写设备，那么对于设备的写入会缓存到 Cache 中。这是一种**错误**的行为，在实际编写代码的时候这么做会引发不可预知的问题。请思考：这么做这会引发什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存更新的策略来考虑。 ■

错误原因：

当外设产生中断信号或更新数据的时候，Cache 中之前旧的数据可能刚完成缓存，那么完成缓存的这部分数据无法完成更新，则会产生错误的行为

区别：

对于串口设备而言，读写频繁，信号多，在相同时间内发生错误的概率远高于 IDE 磁盘

5.2

> Thinking 5.2　查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？一个目录下最多能有多少个文件？我们的文件系统支持的单个文件最大为多大？ ■

```
char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
```

在文件控制块结构体中，存在 f_pad 属性，大小为 BY2FILE 即为 256B，表示一个文件控制块大小为 256B，那么一个 4KB 大小的磁盘块最多可以存储 16 个文件控制块

目录块即为整个块都用来存放文件目录项的磁盘块，一个目录项大小为 4B（和页表项大小相同），一个目录块大小为 4KB，则一个目录中可以存放 1024 个目录项，又因为一个目录项指向一个 4KB 大小的数据块，而一个数据块中可以存放 16 个文件控制块，因此一个目录下可以存放 16384K 个文件控制块

一个文件块中有直接和间接指针加起来一共 1024 个，每个指针指向一个磁盘块，每个磁盘块的大小都为 4KB，因此单个文件的大小为 1024 × 4KB = 4MB

注意：这里我们模拟的是索引节点的形式，也就是有 inode 支持的文件结构，因此目录项会先所引到 inode 节点，也就是上面我们分析的存放着**16个FCB**的那个磁盘块，这里它的作用就类似于索引节点，根据这个索引节点，我们可以找到真正的文件数据块

5.3

> Thinking 5.3　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？ ■

操作系统把 DISKMAP 到 DISKMAP + DISKMAX 这一段虚拟地址空间作为缓冲区，DISKMAX = 0x40000000，最多处理 1GB 的空间

5.4

> Thinking 5.4　在本实验中，`fs/serv.h`、`user/include/fs.h` 等文件中出现了许多宏定义，试列举你认为较为重要的宏定义，同时进行解释，并描述其主要应用之处。 ■

lab5 一大特点就是引进了很多宏定义，他们主要记录在 serv.h，fs.h，fd.h，fsreq.h 这四个头文件中，下面会依次进行解析

fs / serv.h

```
#define PTE_DIRTY 0x0004 // 页表项 dirty 标志位
#define SECT_SIZE 512 // 一个磁盘扇区的大小为 512B
#define SECT2BLK (BLOCK_SIZE / SECT_SIZE) // 一个磁盘块中的扇区数
#define DISKMAP 0x10000000 // 磁盘块在内存中映射的起始地址
#define DISKMAX 0x40000000 // 用于磁盘块映射的地址空间

/* ide.c - 底层磁盘的 I/O 接口 */

void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);// 从编号为 diskno 的磁盘块中，读取 nsecs 个扇区，到 dst 所指向的内存中(这里的内存是指 kseg1 缓冲区中 DISKMAP ~ DISKMAP+DISKMAX 范围)，起始扇区号为 secno
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);// 向编号为 diskno 的磁盘块中，写入 nsecs 个扇区，数据来源为 src 所指向的内存地址，起始扇区号为 secno

/* fs.c - 文件系统核心接口 */
```

```
13
14 int file_open(char *path, struct File **pfile);// 打开 path 路径所指向的文件，把文件控制块存入 pfile
15 int file_create(char *path, struct File **file);// 按照 path 路径创建文件，把文件控制块存入 file
16 int file_get_block(struct File *f, u_int blockno, void **pblk);// 获取文件控制块 f 表示的文件，第 block 个逻辑块(在文件内部的分块)对应的磁盘块的内存地址(块缓存)，把这个内存地址存入 pblk 中
17 int file_set_size(struct File *f, u_int newsize);// 设置文件控制块 f 表示的文件的大小为 newsize
18 void file_close(struct File *f);// 关闭 f 所代表的文件
19 int file_remove(char *path);// 删除 path 路径下的文件(注意只删最后一级)
20 int file_dirty(struct File *f, u_int offset);// 设置 f 所代表的文件，偏移量为 offset 的文件块的 dirty 标志位，以便后续写回磁盘
21 void file_flush(struct File *f);// 把文件中所有的脏块都同步写回磁盘
22 int block_is_free(u_int blockno);// 检查 blockno 对应的磁盘块是否空闲
23
24 /* 文件系统初始化与块管理 */
25 void fs_init(void);// 初始化文件系统(加载 super，bitmap，根目录)，必须在一切 fs_* 函数之前执行
26 void fs_sync(void);// 把整个文件系统都同步回磁盘(全局级别的 flush)
27 extern uint32_t *bitmap;// 指向磁盘块的位图
28 int map_block(u_int blockno);// 把第 blockno 个磁盘块映射到内存(DISKMAP 区域)，以供直接访问
29 int alloc_block(void);// 在 bitmap 中寻找一个空闲的磁盘块并且分配，返回块号
```

所有有返回值的函数，未特殊说明时，都是操作成功返回 0，不成功就返回负的错误码

user / include / fs.h

```
1  #define BLOCK_SIZE PAGE_SIZE // 文件系统中一个块的大小(4KB)
2  #define BLOCK_SIZE_BIT (BLOCK_SIZE * 8) // 一个块中的 bit 位数
3  #define MAXNAMELEN 128 // 文件名(单个目录)的最大长度，包含结尾\0
4  #define MAXPATHLEN 1024 // 完整路径最大长度，包括null
5  #define NDIRECT 10 // 文件控制块中直接指针的数量
6  #define NINDIRECT (BLOCK_SIZE / 4) // 一个间接块中的指针数量(一个指针为 4B，间接块就是全都存指针的磁盘块)
7  #define MAXFILESIZE (NINDIRECT * BLOCK_SIZE) // 单个文件最大大小
8  #define FILE_STRUCT_SIZE 256 // 每个文件控制块的大小(类似于索引节点)
9
10 struct File {
11         char f_name[MAXNAMELEN]; // 文件名
12         uint32_t f_size;         // 文件大小
13         uint32_t f_type;         // 文件类型
14         uint32_t f_direct[NDIRECT]; // 直接指针个数
15         uint32_t f_indirect;        // 间接指针
16         struct File *f_dir; // 指向上级目录文件控制块的指针
17         char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 -
   sizeof(void *)];
18 } __attribute__((aligned(4), packed));
19
20 #define FILE2BLK (BLOCK_SIZE / sizeof(struct File))// 每个磁盘块中能存放的文件控制块的个数(通常为 16)
21 #define FTYPE_REG 0 // 普通文件
22 #define FTYPE_DIR 1 // 目录
23 #define FS_MAGIC 0x68286097 // 魔数
24 struct Super {
```

```
        uint32_t s_magic;   // 验证文件系统的魔数
        uint32_t s_nblocks; // 磁盘块的总数量
        struct File s_root; // 根目录的文件控制块
};
```

user / include / fd.h

```
#define MAXFD 32 // 最大文件描述符数量(Fd 的数量)
#define FILEBASE 0x60000000 // 文件映射区的基地址，内核或用户空间中用于文件数据映射的起始虚拟地址
#define FDTABLE (FILEBASE - PDMAP) // 文件描述符表的基地址，计算方法是用 FILEBASE 减去 PDMAP(通常是4MB，这部分空间是预留给 FDTABLE 的空间，确保两部分不会重叠)
#define INDEX2FD(i) (FDTABLE + (i) * PTMAP) // 给定文件描述符索引 i，计算该文件描述符对应的虚拟地址
#define INDEX2DATA(i) (FILEBASE + (i) * PDMAP) // 给定文件描述符索引 i，计算文件数据映射区中对应数据块的虚拟地址

struct Fd;
struct Stat;
struct Dev;

// 设备抽象结构体
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);
};

// 文件描述符结构体
struct Fd {
        u_int fd_dev_id;
        u_int fd_offset;
        u_int fd_omode;
};

// 文件状态结构体
struct Stat {
        char st_name[MAXNAMELEN];
        u_int st_size;
        u_int st_isdir;
        struct Dev *st_dev;
};

// 文件描述符与文件数据的复合体
struct Filefd {
        struct Fd f_fd;
        u_int f_fileid;
        struct File f_file;
};

int fd_alloc(struct Fd **fd); // 分配一个新的文件描述符，并且把描述符存入 fd 指针中
int fd_lookup(int fdnum, struct Fd **fd); // 根据给出的文件描述符编号，查找对应的 struct Fd 结构体(有合法性检查)
```

```
void *fd2data(struct Fd *); // 根据给定的文件描述符结构体指针，查找对应的数据区内存地址(实际上是文件数据缓冲区的虚拟地址)
int fd2num(struct Fd *); // 根据给出的文件描述符结构体指针，返回对应的编号
int dev_lookup(int dev_id, struct Dev **dev); // 根据设备 id 查找对应的设备结构体 struct Dev，查找结果存入 dev 中
int num2fd(int fd); // 根据文件描述符编号 id 查找文件描述符结构体指针(没有合法性检查)
extern struct Dev devcons;
extern struct Dev devfile;
extern struct Dev devpipe;
```

user / include / fsreq.h

```
// 请求类型编号，每种请求类型对应一个结构体用以传递参数
enum {
    FSREQ_OPEN,     // 打开文件请求
    FSREQ_MAP,      // 映射文件某一页到内存
    FSREQ_SET_SIZE, // 设置文件大小
    FSREQ_CLOSE,    // 关闭文件
    FSREQ_DIRTY,    // 标记文件某一页为“脏页”
    FSREQ_REMOVE,   // 删除文件
    FSREQ_SYNC,     // 同步文件系统
    MAX_FSREQNO     // 最大请求编号(不是请求类型，只是用于判断)
};

// 打开文件的请求
struct Fsreq_open {
    char req_path[MAXPATHLEN];  // 文件路径
    u_int req_omode;            // 打开模式（如 O_RDWR, O_CREAT 等）
};

// 把文件的某一页映射到内存中的请求
struct Fsreq_map {
    int req_fileid;     // 要映射的文件 ID（由打开返回）
    u_int req_offset;   // 映射偏移（以页为单位）
};

// 调整文件大小的请求
struct Fsreq_set_size {
    int req_fileid;     // 文件 ID
    u_int req_size;     // 新的文件大小
};

// 关闭文件的请求
struct Fsreq_close {
    int req_fileid;     // 要关闭的文件 ID
};

// 告诉文件系统某一页已经被修改，可能需要写回磁盘
struct Fsreq_dirty {
    int req_fileid;     // 文件 ID
    u_int req_offset;   // 标记脏页的偏移
};

// 删除某个路径下的文件的请求
struct Fsreq_remove {
    char req_path[MAXPATHLEN];  // 要删除的文件路径
};
```

5.5

> **Thinking 5.5** 在 Lab4“系统调用与 fork”的实验中我们实现了极为重要的 **fork** 函数。那么 fork 前后的父子进程是否会共享文件描述符和定位指针呢？请在完成上述练习的基础上编写一个程序进行验证。 ■

fork 前后父子程序共享文件描述符和定位指针

验证程序：

```
#include "lib.h"

static char *msg = "This is the NEW message of the day!\n\n";
static char *diff_msg = "This is a different massage of the day!\n\n";

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

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

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

        int id;

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

```
47              }
48              writef("father read is good && father_fd == %d\n",r);
49              struct Fd *fdd;
50              fd_lookup(r,&fdd);
51              writef("father_fd's offset == %d\n",fdd->fd_offset);
52          }
53  }
```

5.6

> **Thinking 5.6** 请解释 **File, Fd, Filefd** 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。 ■

具体解释详见**文件顶层函数**部分

5.7

> **Thinking 5.7** 图 5.9 中有多种不同形式的箭头，请解释这些不同箭头的差别，并思考我们的操作系统是如何实现对应类型的进程间通信的。 ■

ENV_CREATE(user_env) 和 ENV_CREATE(fs_serv) 都是**异步消息**，init() 函数在发出创建消息后， 可以自行返回执行后续步骤，由 fs 和 user 线程自己完成相应的初始化工作

fs 线程完成 serv_init() 和 fs_init() 的初始化以后，进程进入 serv() 函数，被 ipc_receive() 阻塞，直到收到 user 线程的 ipc_send(fsreq) 被唤醒，后继续执行

user 线程向 fs 线程 ipc_send(fsreq) 发送请求为同步消息，发送后自身进入阻塞，等待被唤醒的 fs 线程服务结束时 ipc_send(dst_va)，用户线程接收到数据后继续运行，此后 fs 线程进入阻塞，等待下次被用户唤醒

课上考试

时间查询与计算

题目要求：

偏移	效果	数据位宽
`0x0000`	读/写：触发时钟更新	4 bytes
`0x0010`	读：从格林威治时间 1970 年 01 月 01 日 00 时 00 分 00 秒起至现在的总秒数	4 bytes
`0x0020`	读：为精确 Unix 时间戳而记录的微秒数，范围为 0 到 999999 微秒（1秒内）	4 bytes

本题需要完成两个用户态中的函数，并要求在 `user/include/lib.h` 中声明，`user/lib/ipc.c` 中实现

1. `u_int get_time(u_int *us)`

从 `rtc` （实时时钟）中读取**总秒数**和**微秒数**，总秒数作为该函数的返回值，微秒数通过指针参数 `us` 返回

1. `void usleep(u_int us)`

在任一用户态进程进入 `usleep` 函数后，必须停留 `us` 微秒后才能退出函数，这其中允许时间片的轮转，也就是说**在等待调度的过程中仍然计时**

- 请避免单位换算时可能发生的整数溢出

- `u_int` 为无符号整数，如果需要进行减法得到有符号整数，请在**运算前**使用 `(u_int)` 进行强制类型转换
- 课程组给出的一种 `usleep` 的实现：

```
u_int get_time(u_int *us) { return syscall_get_time(us); }

void usleep(u_int us) {
  u_int inUs = 0;// 开始时的微秒
  u_int nowUs = 0;// 当前的微秒
// 调用一次 get_time 函数同时获得开始记录时的 inUs 和 inTime
  u_int inTime = get_time(&inUs);

  while (1) {
    // 每次进入循环，都获得当前的 nowTime 和 nowUs
    u_int nowTime = get_time(&nowUs);
    int del = (int)nowTime - (int)inTime;
    // 当前时间 - 开始记录的时间 -> 统一换算成微秒考虑，和 us 进行比较，超过时就直接返回，反之就进行调度，调度过程中仍然要记录时间延迟
    if (del * 1000000 + ((int)nowUs - (int)inUs) > us) {
      return;
    } else {
      syscall_yield();
    }
  }
}
```

同时需要增加一个系统调用，用来实现 get_time 的内核函数实现，在**秒**存入 time 寄存器的同时，把**微妙**存入 us 寄存器：

```
// 函数功能 : 从特定内存地址读取当前信息
u_int sys_get_time(u_int *us) {
  u_int time = 0;
// 注意，这里的刷新不是初始化，只是为了把 RTC 中当前时钟的时间存入time 寄存器内部，防止出现读秒的时候寄存器还没更新的情况，相当于先按住读秒操作，强行把更新后的时间准备好
  memcpy(&time, (KSEG1 | 0x15000000), sizeof(u_int));
  memcpy(&time, (KSEG1 | 0x15000010), sizeof(u_int));  // store time in 'time'
  memcpy(us, (KSEG1 | 0x15000020), sizeof(u_int));     // store utime in 'us'

  return time;
}
```

值得注意的是，学长这个系统调用函数的实现非常巧妙，它相当于实现了 ioread 系列函数的平替，我们先回顾 ioread 系列函数的作用：

```
// 入参 paddr 是内核想访问的那个设备寄存器所在的物理地址
static inline uint8_t ioread8(u_long paddr) {
        return *(volatile uint8_t *)(paddr | KSEG1);
}
```

看似反常的“ 把物理地址映射成虚拟地址 ”的过程：

物理地址映射成虚拟地址是操作系统或内核在实现访问设备（如 MMIO 外设寄存器）时的特殊处理

设备寄存器往往是固定映射在物理地址空间的某个范围内，**它们没有对应的正常的虚拟地址映射**，或者**内核默认没有映射它们的虚拟地址**

而内核是没法直接访问物理地址的，也要通过访问虚拟地址来间接访问，为了让内核代码能用“虚拟地址”的方式访问设备寄存器，需要**人为建立一段虚拟地址空间，映射到设备的物理地址**，这就是“映射物理地址到虚拟地址”

现在我们知道这么做很**反常**，那么能不能不经过这个转化，直接访问虚拟地址？这就是上文代码中给出的思路：

```
memcpy(&time, (KSEG1 | 0x15000000), sizeof(u_int));
```

其中地址 15000000 是设备寄存器所在的物理地址，我们现在知道了调用 ioread 就是把这个物理地址转化成虚拟地址再进行访问的，那么我们可以手动进行 + KSEG1 的转化，这么做我们就省去了在调用 sys_read_dev 的时候先传入**目标物理地址**（由用户顶层给出），再调用 ioread 进行虚拟地址转换的多此一举，是一个很巧妙的设计

文件请求路径

下面的例题本质上考察的就是如何逐步添加一个新的文件请求，可以类比 lab4 中有关系统调用添加的过程

Lab5 的课下内容中，我们实现了通过**绝对路径**，也即相对于磁盘根目录而言的路径，打开文件，为其获取 `struct Open` 与 `struct File` 的函数—— `file.c / open`

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

现在需要我们拓展打开文件的形式，额外实现一个通过**指定目录 + 相对路径**来打开文件的函数 `file.c / openat`

```
int openat(int dirfd, const char *path, int mode);
```

在这里，我们保证：

- 目录代表的文件已经在**文件管理系统**中（拥有 `Open`）与**当前进程**中（拥有 `Fd`）被打开，即可以通过其 `fdnum` 对应的 `struct Fd` 获取信息
- 调用 `open`、`openat` 的所有目录 / 文件均存在，且 `openat` 只会打开普通文件（非目录）
- 调用 `openat` 打开的路径均为相对路径，即不会以 `/` 开头

实现过程

我们可以接着 file.c 的内容向下层实现，也就是从 fsipc.c 开始逐层实现后面的逻辑

1. `user / include / fsreq.h`：增加一个对于文件服务进程的请求类型 `FSREQ_OPENAT` 和请求结构体 `struct Fsreq_openat`，完成基本的数据结构准备
2. `user / lib / file.c`：仿照 `open` 函数实现 `openat` 函数，设计该函数对用户进程的顶层接口
3. `user / lib / fsipc.c`：仿照 `fsipc_open` 实现 `fsipc_openat` 函数，完成对 `Fsreq_openat` 各个字段的赋值，使得其能够发送 `openat` 的请求
4. `fs / serv.c`：
    - 修改 `srev` 函数，使其能够转发 `FSREQ_OPENAT` 的请求
    - 仿照 `serv_open` 函数实现 `serve_openat` 函数，具体实现 `openat` 函数的整体功能
5. `fs / fs.c`：

- 仿照 `walk_path` 实现 `walk_path_at`，从**文件层面**实现按 `path` 路径查找文件的功能
- 仿照 `file_open` 实现 `file_openat`，类似调用 `walk_path_at` 函数，封装功能，供文件服务进程调用

6. 头文件：
   - `user / include / lib.h`：增加 `openat`，`fsipc_openat` 声明
   - `fs / serv.h`：增加 `file_openat` 的声明

过程中仍然保留了 `openat` 字样，实际可以根据情况自由调整。除了 `fs.c` 中功能具体实现的方式不同之外，基本上面几个文件的修改都是有规律的，和 Lab4 添加系统调用类似

具体代码层面

user / include / fsreq.h

增加一个新的文件请求类型 Fsreq_openat，这里可以仿照 Psreq_open 的形式实现，注意需要同时添加**结构体**和**操作编号**的宏定义

```
struct Fsreq_openat {
    u_int dir_fileid;           // 相对的 '根目录' 打开的文件 id
    char req_path[MAXPATHLEN]; // 打开的文件的相对路径
    u_int req_omode;            // 打开文件的模式
};
#define FSREQ_OPENAT 8
```

其他声明：

需要在 `user/include/lib.h` 中增加 `int openat(int dirfd, const char *path, int mode)` 、`int fsipc_openat(u_int, const char *, u_int, struct Fd *)` 函数声明

在 `fs/serv.h` 中增加 `int file_openat(struct File *dir, char *path, struct File **pfile)` 函数声明

user / lib / file.c

仿照 open 函数实现 int openat(int dirfd, const char *path, int mode)，实现这一函数的相关提示：

- 调用 fd_lookup 利用 dirfd 查找 dirfd 的文件描述符 struct Fd *dir
- 将 struct Fd *dir 指向的类型转换为 struct Filefd 后获得 dirfd 对应的 fileid
- 调用 fsipc_openat 打开文件并完成对指针 fd 的赋值

```
// 函数功能 : 在文件系统的相对目录 dirfd 下，打开 path 路径对应的文件，打开方式为 mode，成功时返回文件描述符 fd(整型)
int openat(int dirfd, const char *path, int mode) {
  struct Fd *dir;
  struct Fd *filefd;
  int r;
// 根据相对目录的文件描述符编号，得到对应的文件描述符结构体本身(调用 fd_lookup)
  fd_lookup(dirfd, &dir);
  if ((r = fd_alloc(&filefd)) < 0) {
    return r;
  }

// 根据相对目录的 Fd 文件描述符结构体，找到其在文件服务进程的文件 ID，这里一定支持强转，因为 dir 文件之前被打开过，相应字段以填写完毕
  int dir_fileid = ((struct Filefd *)dir)->f_fileid;
  // 调用 fsipc_openat 函数，入参含义分别为目录文件在文件服务进程中的 ID，目标文件相对路径，打开方式，待填写的打开文件结构体 filefd
```

```
  if ((r = fsipc_openat(dir_fileid, path, mode, filefd)) < 0) {
    return r;
  }

  char *va;
  struct Filefd *ffd;
  u_int size, fileid;

// 根据打开文件的 fd 得到打开文件对应的参数 : 缓冲区虚拟地址，filefd 结构体(重新存一份)，文件大小，文件 ID
  va = fd2data(filefd);
  ffd = (struct Filefd *)filefd;
  size = ffd->f_file.f_size;
  fileid = ffd->f_fileid;

// 把文件内容映射回缓冲区，方便后续访问
  for (int i = 0; i < size; i += BY2BLK) {
    // 调用 fsipc_map，入参含义 : 把文件 ID 为 fileid 的文件，偏移为 i 位置出，大小为 4KB 的内容，映射到虚拟地址 va + i 的位置
    if ((r = fsipc_map(fileid, i, (void *)(va + i))) != 0) { return r; }
  }

// 最后返回的是文件的 fd 编号
  return fd2num(filefd);
}
```

user / lib / fsipc.c

仿照 fsipc_open 实现 int fsipc_openat(u_int dir_fileid, const char *path, u_int omode, struct Fd *fd)，完成对 Fsreq_openat 各个字段的赋值，并与文件系统服务进程进行通信

```
int fsipc_openat(u_int dir_fileid, const char *path, u_int omode, struct Fd *fd) {
    struct Fsreq_openat *req;
    u_int perm;
// 类型强转 : 告诉系统请把 fsipcbuf 当成是 Openat 类请求对待
    req = (struct Fsreq_openat *)fsipcbuf;
    if (strlen(path) >= MAXPATHLEN) {
        return -E_BAD_PATH;
    }

// 设置操作参数 : 文件打开方式，上级目录的文件控制块，待打开文件的路径
    req->req_omode = omode;
    req->dir_fileid = dir_fileid;
    strcpy((char *)req->req_path, path);

// 核心操作 : 把文件打开请求发送给文件服务进程，打开结果返回给用户进程，这里用户进程希望文件服务系统返回打开文件的文件描述符，因此需要传 fd 作为 dstva 参数
    return fsipc(FSREQ_OPENAT, req, fd, &perm);
}
```

fs / serv.c

仿照 serve_open 实现 serve_openat 函数，并在 serve 函数中增加关于 openat 请求的判断

```
void serve_openat(u_int envid, struct Fsreq_openat *rq) {
    struct Open *pOpen;
```

```
    struct Open *o;
    struct File *f;
    struct Filefd *ff;
    int r;

// 查找 envid 这个进程具体的打开文件列表，找到待创建文件的上级目录 dir 对应的 Open 结构体
    if ((r = open_lookup(envid, rq->dir_fileid, &pOpen)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
    struct File *dir = pOpen->o_file;

// 调用具体的 file_openat 函数，在该目录下打开目标路径的文件
    if ((r = file_openat(dir, rq->req_path, &f)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }

// 为了记录文件被打开，需要分配一个 Open 表项
    if ((r = open_alloc(&o)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }

// 设置返回的文件描述符结构
    o->o_file = f;// Open 结构体绑定刚刚打开的文件

    ff = (struct Filefd *)o->o_ff;// o->o_ff 是共享给用户的页，这里转换为 struct Filefd 指针，用于写入信息
    ff->f_file = *f;// 将文件信息复制给用户，注意是复制，不是指针共享
    ff->f_fileid = o->o_fileid;// 把文件 ID 传回去，用于后续操作
    o->o_mode = rq->req_omode;// 记录打开模式
    ff->f_fd.fd_omode = o->o_mode;// 设置文件描述符中的模式和设备类型
    ff->f_fd.fd_dev_id = devfile.dev_id;// 设置设备类型

// 通过 IPC 返回文件描述符所在的页面给用户进程
    ipc_send(envid, 0, o->o_ff, PTE_D | PTE_LIBRARY);
}
```

fs / fs.c

仿照 walk_path 实现 `int walk_path_at(struct File *par_dir, char *path, struct File **pdir, struct File **pfile, char *lastelem);`

在 par_dir 目录下按相对路径 path 查找文件，并仿照 file_open 实现 `int file_openat(struct File *dir, char *path, struct File **file)` 调用 walk_path_at 函数

注意相较于原有的函数，新实现的函数需要传入相对路径根目录的文件块，一路传到最底层的 walk_path_at 函数中，这个相对路径根目录是用来代替 super->s_root 的作用的

```
// 这个函数完全按照 walk_path 来实现即可
// 入参含义 : 相对路径根目录文件控制块，文件路径，待返回的上级目录文件控制块，目标文件的文件控制块，如果找不到目标文件返回的路径中的最后一个目录
int walk_path_at(struct File *par_dir, char *path, struct File **pdir,
                 struct File **pfile, char *lastelem) {
  char *p;
```

```
  char name[MAXNAMELEN];
  struct File *dir, *file;
  int r;

// 这里只有文件路径的初始化需要修改，首先 file 原本记录的是根目录下第一级目录的文件控制块，这里我们直接记成相对根目录的控制块即可；其次 path 已经是一个相对目录，不需要再跳过相对目录，直接使用即可，也就是下面这两条代码不再需要
  /*
    path = skip_slash(path);
    file = &super->s_root;
  */
  file = par_dir;
  dir = 0;
  name[0] = 0;

  if (pdir) {
    *pdir = 0;
  }

  *pfile = 0;

  while (*path != '\0') {
    dir = file;
    p = path;

    while (*path != '/' && *path != '\0') { path++; }

    if (path - p >= MAXNAMELEN) { return -E_BAD_PATH; }

    memcpy(name, p, path - p);
    name[path - p] = '\0';
    path = skip_slash(path);
    if (dir->f_type != FTYPE_DIR) { return -E_NOT_FOUND; }

    if ((r = dir_lookup(dir, name, &file)) < 0) {
      if (r == -E_NOT_FOUND && *path == '\0') {
        if (pdir) { *pdir = dir; }

        if (lastelem) { strcpy(lastelem, name); }

        *pfile = 0;
      }

      return r;
    }
  }

  if (pdir) { *pdir = dir; }

  *pfile = file;
  return 0;
}

int file_openat(struct File *dir, char *path, struct File **file) {
  return walk_path_at(dir, path, 0, file, 0);
}
```

这个题考察了如何添加新的文件服务请求的种类，这个信息的流动通路是文件系统中的很重要一环，主要是这个考点扩展性很强，随便换个操作，也只是各个文件中的具体会改变，其余路径上的信息流动和五个文件的访问顺序几乎都是大同小异的

SSD 硬盘模拟

题目背景

在 MOS 的实现中，我们使用的是 GXemul 提供的 IDE 磁盘作为外挂的文件存储，现在我们使用固态硬盘 SSD 的概念对 IDE 磁盘进行模拟，即按照操作 SSD 的方法去操作我们的磁盘

SSD 的特点是每个块存储数据后必须**先擦除才能写入**，同时闪存块是有擦写次数上限的，为了平衡各个块的擦写次数，SSD 引入了**磨损平衡**机制，我们在本次模拟过程中规定，模拟 SSD 的**读、写、擦除**均以块为单位，每块的大小都是 512B；且对于每个物理块，必须**先擦除，才能写入数据**

我们定义如下几个概念：

逻辑块号：文件系统读写的逻辑硬盘块编号，在物理上不一定连续，每个块的大小为512B

物理块号：SSD 实际读写的闪存的物理编号，在物理上连续，每个块的大小都是512B

闪存映射表：记录逻辑块号对应物理块号信息的表

物理块位图：记录各个物理块的状态信息

实际上，我们要先实现一个逻辑硬盘块到物理硬盘块的映射表，同时维护每个物理块的使用情况与擦除次数，并根据这些信息模拟正常的 SSD 工作流程

首先说明，外界访问 SSD 时使用的是其逻辑块号，我们需要经**闪存映射表**获得其物理块号后，在 IDE 磁盘的对应**扇区**执行相关操作

我们先规定 SSD 硬盘可能出现的四种操作：

1. **初始化硬盘**：清空闪存映射表，在物理块位图中将 SSD 的所有物理块初始化为可写状态，SSD各物理块的累计擦写次数清 0（保证只在开始时调用一次）
2. **读取逻辑块**：查询闪存映射表，找到逻辑块号对应的表项。如果为空，则返回 `-1`；若不为空，就去读取对应的物理块，返回物理块内的数据，并返回 `0`
3. **写入逻辑块**：查询闪存映射表，找到逻辑块号对应的表项，若为空（初次写），则分配一个可写的物理块，在闪存映射表中填入此映射关系，将数据写入该物理块，并标记此物理块为不可写；若不为空（覆盖写），则擦除原来的物理块并清除该逻辑块号的映射，并按相同方法分配一个物理块向其写入数据。
4. **擦除逻辑块**：查询闪存映射表，找到逻辑块号对应的表项，若为空，不做处理；若不为空，则将对应的物理块擦除（擦除表示将物理块数据全部清 0，下同）并清除此映射关系

为了描述上面需要的状态，我们需要在代码中定义如下数据结构：

- **闪存映射表**：记录逻辑块号对应物理块号信息的表
- **物理块位图**：记录各个物理块是否可写。每次向可写的物理块中写入数据后，都需要标记物理块为不可写
- **物理块累计擦除次数表**：记录各个物理块的累计擦除次数

在完成上述基础机制后，我们还需要实现一个简单实现磨损平衡的分配函数

题目要求

在本题中，你需要使用 IDE 0 号硬盘（`diskno = 0`）的第 0 ~ 31 个扇区来模拟SSD的第 0~31 个物理块，之后根据题目描述部分的描述，在 `fs / ide.c` 中实现下列函数

1. 初始化硬盘

```
void ssd_init();
```

- 功能：初始化闪存映射表、物理块位图、物理块累计擦除次数表。

2. 读取逻辑块

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

- `logic_no`：逻辑块号
- `dst`：硬盘数据要加载到的内存地址
- 功能：读取逻辑块 `logic_no` 的数据到 `dst`，成功返回 `0`，失败返回 `-1`

3. 写入逻辑块

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

- `logic_no`：逻辑块号
- `src`：要写入到硬盘中的数据的起始内存地址
- 功能：向逻辑块号对应的硬盘块写入以 `src` 为起始地址的512字节数据

4. 擦除逻辑块

```
void ssd_erase(u_int logic_no);
```

- `logic_no`：逻辑块号
- 功能：擦除逻辑块 `logic_no` 上的数据，并取消 `logic_no` 到物理块号的映射关系

评测时将会调用上面的接口做一系列操作，验证操作执行结果是否正确，并会在操作的任何阶段读取硬盘数据以检查实现的正确与否

为了实现上面的函数，你可能还需要自行实现以下两个函数，但我们不对函数的定义做具体要求：

- 擦除物理块：将全 0 的数据写入到物理块中，物理块累计擦除次数加一，并置为可写状态
- 分配新物理块：按"题目描述"部分的方法分配，需要用到物理块位图和物理块累计擦除次数两个信息

具体实现

需要的数据结构：

```
u_int ssd_bitmap[32]; // 第 i 个物理块是否可用，1 - 可用(严格来讲不是位图，就是一个简单的下标数组)
u_int ssd_clean_up[32]; // 第 i 个物理块的擦除次数
u_int ssd_map[32];  // 逻辑块号 i 映射到的物理块号
```

初始化函数

```
void ssd_init() {
  for (int ssdno = 0; ssdno < 32; ssdno++) {
    ssd_bitmap[ssdno] = 1;
    ssd_cleanup[ssdno] = 0;
    ssd_map[ssdno] = (u_int)0xffffffff;
  }
}
```

这里 SSD 磁盘块的编号是从 0 开始的，也就是说逻辑块号有可能是 0，那么我们的映射表不存在映射的时候就不能存 0，可以存一个 INF，防止系统误认为第 0 个逻辑块发生了映射

ssd_read

因为我们使用 ide 磁盘的第 0 块实现的 ssd_read，所以真正的读功能本质上是由 ide_read 来实现的，下面我们先回顾一下我们曾经实现过的两个硬盘驱动接口函数：

```
// 读取扇区 : 从第 diskno 个磁盘的第 secno 个扇区开始，读取 nsecs 个扇区的数据，存储到内存地址 dst 中(dst 是物理内存在进程虚拟地址空间中的映射)
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);
// 写入扇区 : 把内存地址 src 处(src 是物理内存在进程虚拟地址空间中的映射)，大小为 nsecs 个扇区的数据，写入第 diskno 个磁盘的第 secno 个扇区
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);
```

如果我们想直接实现对磁盘块的底层读写，可以调用这两个函数来实现

```
int ssd_read(u_int logic_no, void *dst) {
  int physics_no = ssd_map[logic_no];
  if (physics_no == (u_int)0xffffffff) return -1;
  ide_read(0, physics_no, dst, 1);
  return 0;
}
```

ide_read 的入参含义：

- 0：题目要求，我们使用的是第 0 号 IDE 磁盘
- physics_no：我们要操作的逻辑块号对应的磁盘物理块号，也就是对应的扇区号（**注意**：这里我们设计的 OS 对此进行了简化，一个磁盘块大小就是一个扇区大小，因此磁盘块号和扇区号可以一一对应）
- dst：读得数据写入的缓存地址（由入参给出）
- 1：题目要求，每次只读 512B，恰好是一个扇区 / 磁盘块的大小

ssd_erase

擦除函数用来消除原有的磁盘映射，注意在擦除的时候，我们需要全面考虑待修改的数据结构：

- 对应物理块号的擦除次数 + 1
- 对应物理块号重新设为可用
- 对应逻辑块号的映射重新设为 0xffffffff
- 注意在磁盘块中**真正**完成擦除（也就是写入**空数据**），这里我们仿照 ssd_read，可以调用 ide_write 来完成对磁盘块的写入操作

```
void ssd_erase(u_int logic_no) {
  if (ssd_map[logic_no] == (u_int)0xffffffff)return;
  physics_no = ssd_map[logic_no];
  ssd_map[logic_no] = (u_int)0xffffffff;
  ssd_clearmap[physics_no]++;
  ssd_bitmap[physics_no] = 1;
  char buf[512] = {0};
  ide_write(0, physics_no, buf, 1);
  return;
}
```

physics_alloc

磨损平衡算法：每次操作我们都需要按照**低负荷状态**和**高负荷状态**两种情况进行处理

1. 低负荷状态：优先选择擦除次数最少的，当前状态可用的磁盘号（擦除次数相同就选择较小编号），如果选出的磁盘号的擦除次数小于 5 次，说明此时处于低负荷状态，直接返回这个磁盘块即可
2. 高负荷状态：如果当前选出的可用磁盘块的最小擦除次数都大于等于 5，此时需要考虑非可用块，先按照 1 中的逻辑，从非可用快中找到擦除次数最少（相同则为编号最小）的块，进行内容的替换操作

将选出的非可用块包含的内容转存到选出的可用块中，这也包含**将映射到该物理块的闪存表项同时更改**；同时擦除非可用块，并进行分配

```
u_int physics_alloc() {
  u_int low_physics = 0x3f3f3f3f;
  u_int high_physics = 0x3f3f3f3f;
  int clear_count = 0x3f3f3f3f;
  // 先找低负荷状态
  for (int i = 0; i < 32; i++) {
    if (ssd_clearmap[i] < clear_count) {
      if (ssd_bitmap[i] == 1) {
        clear_count = clear_map[i];
        low_physics = i;
      }
    }
  }
  if (clear_count < 5) {
    return low_state;
  }
  // 低负荷状态不满足，再找高负荷状态
  for (int i = 0; i < 32; i++) {
    if (ssd_clearmap[i] < clear_count) {
      if (ssd_bitmap[i] == 0) {
        clear_count = clear_map[i];
        high_physics = i;
      }
    }
  }
  // 高负荷状态写入低负荷状态
  char buf[512] = {0};
  ide_read(0, high_physics, buf, 1);
  ide_write(0, low_physics, buf, 1);
  ssd_bitmap[low_physics] = 0;
  // 修改高负荷块的映射关系，他现在真正的内容在低负荷块中
  for (int i = 0; i < 32; i++) {
    if (ssd_map[i] == high_physics) {
      ssd_map[i] = low_physics;
      break;
    }
  }
  // 高负荷状态内容擦除，这里不需要重新设置为可用，因为紧接着我们就要向其中写入新的内容
  memset(buf, 0, sizeof buf);
  ide_write(0, high_physics, buf, 1);
  ssd_clearmap[high_physics]++;
  // 返回高负荷状态
  return high_physics;
}
```

注意这里很容易遗漏的一步是，高负载块的内容写入低负载块的时候，需要重新建立虚实块号之间的映射，把原本映射到高负载块的逻辑块号修改到低负载块，否则我们就没法索引到原有的高负载块的内容了（相当于无法找到一个逻辑块号能映射到低负载块，也就找不到原有高负载块的内容）

ssd_write

在实现了平衡擦除策略后，我们可以封装好分配物理块的操作，每次尝试写入的时候，physics_alloc 得到的新物理块一定是合法的

如果 logic_no 之前没有映射过物理块：

1. 分配一个新的物理块
2. 建立映射
3. 该物理块标记为不可用

如果 logic_no 之前映射过物理块：

1. 找到原映射物理块
2. 擦除原有数据

```
void ssd_write(u_int logic_no, void *src) {
  u_int physics_no = (u_int)0xffffffff;
  // 先判断对应物理块是否为空，如果不空要进行擦除后写，如果为空则要申请新物理块，建立映射后写
  // 空
  if (ssd_map[logic_no] == (u_int)0xffffffff) {
    physics_no = physics_alloc();
    ssd_map[logic_no] = physics_no;
    ssd_bitmap[phtsics_no] = 0;
  } else {
    // 不空
    ssd_erase(logic_no);
    physics_no = ssd_map[logic_no];
  }
  ide_write(0, physics_no, src, 1);
  return;
}
```

以上就是所有需要的函数，本例考察的核心还是要**写全**每组操作对我们所维护的数据结构所带来的影响，不要遗漏某些标记或记录的更新

系统时间 & 分条读取

题目要求：

“请你为一个简化的文件系统设计 RAID-0（条带化）读写操作（数据分成奇偶块分别写入磁盘块），同时实现从时间设备读取当前时间的函数”，要求你基于已有的 IDE 读写接口 `ide_read()` 和 `ide_write()` 实现 RAID-0 模式的分条存取，以及从设备地址读取时间信息

time_read

这个函数和 sin 学长的时间查询功能几乎相同，只是最后没有存入给定的 us 寄存器中，同时下面会给出借助 syscall_read_dev 进而调用 ioread 的实现方法，根据上面的分析，如果使用 syscall_read_dev 的话对应的设备地址传入**物理地址**即可

```c
int time_read() {
    int time = 0;
    if (syscall_read_dev((u_int) & time, 0x15000000, 4) < 0)
        user_panic("time_read panic");
    if (syscall_read_dev((u_int) & time, 0x15000010, 4) < 0)
        user_panic("time_read panic");
    return time;
}
```

raid0_write

实现 RAID-0 的写入操作，即把数据**按块**交错写入两块磁盘，这里涉及到磁盘的写入还是底层对磁盘块的操作，使用 ide_write 即可，其中 raid0_write 的入参含义为：待写入磁盘块的起始块号，数据源所在地址，打算写入的块数

```c
void raid0_write(u_int secno, void *src, u_int nsecs) {
    for (i = secno; i < secno + nsecs; i++) {
        if (i % 2 == 0) {
          ide_write(1, i >> 1, src + (i - secno) * 512, 1);
        } else {
           ide_write(2, i >> 1, src + (i - secno) * 512, 1);
        }
    }
}
```

需要注意 ide_write 中写入地址的实现，也就是这行代码：

```c
ide_write(1, i >> 1, src + (i - secno) * 512, 1);
```

其中 i 是当前遍历到的扇区号，secno 是起始的扇区号，那么 i - secno 就是当前这个扇区占所有扇区的偏移量，我们要用这个偏移量去和物理地址中的源数据对其，也就是扇区中每偏移一个扇区号，物理地址就相应偏移一个扇区的大小，也就是 512B，相当于做了一个**写入扇区**和**源数据物理地址**之间的线性映射

同时，因为是要在奇偶盘上分别映射，为了不留空白盘块，每个磁盘上的盘块号 i' 应该是总盘块号 i 的一半

raid0_read

有了 write 操作的实现，read 几乎完全相同，只需要把 ide_write 改成相应的 ide_read 即可

```c
void raid0_read(u_int secno, void *dst, u_int nsecs) {
  for (int i = 0; i < secno; i++) {
    if (i % 2 == 0) {
      ide_read(1, i >> 1, dst + (i - secno) * 512, 1);
    } else {
      ide_read(2, i >> 1, dst + (i - secno) * 512, 1);
    }
  }
}
```

符号链接文件

题目背景

在 MOS 中我们只实现了两种文件类型：常规文件 & 目录文件。现在我们需要实现一种新的文件类型：**符号链接文件**。这类文件会指向一个存在于文件系统中的文件（也可能是符号链接文件），当使用 `open` 访问这些文件时，会等效于访问了他们所指向的文件，类似于 Windows 中的快捷方式

题目约束

符号链接指向的文件一定是使用**绝对路径**表示，并且文件一定存在；

符号链接文件只会指向普通文件和符号链接文件（也就是不会指向目录文件）

- 链接不会出现环
- 文件名长度合法

具体实现

在 `user/include/fs.h` 中，定义新的文件类型

```
// File types
#define FTYPE_LNK 2 // Symbolic link file
```

修改 `tools/fsformat.c`，使得 `fsformat` 工具在写入文件时根据传入的各个文件的类型来决定写入磁盘镜像的文件的类型（`FTYPE_LNK`、`FTYPE_DIR` 或 `FTYPE_REG`），以下是一种修改的思路：

（1）将 `main()` 函数中调用的 `stat()` 函数改为 `lstat()`（参数不用改变），`stat()` 或 `lstat()` 的作用都是将一个文件的相关信息储存在 `statbuf` 结构体中，它们的区别在于：`stat()` 函数会跟踪链接，会解析链接最终指向的文件；而 `lstat()` 不会跟踪链接，直接解析链接文件本身，而我们需要读取链接文件本身的信息，所以需要使用 `lstat()` 函数替换 `stat()` 函数（直接改函数名即可，这两个函数都是 Linux 的库函数）

```
int main(int argc, char **argv) {

  for (int i = 2; i < argc; i++) {
    char *name = argv[i];
    struct stat stat_buf;
    int r = lstat(name, &stat_buf);  // stat 修改在此处，直接变成 lstat 即可
    assert(r == 0);
    if (S_ISDIR(stat_buf.st_mode)) {
      printf("writing directory '%s' recursively into disk\n", name);
      write_directory(&super.s_root, name);
    } else if (S_ISREG(stat_buf.st_mode)) {
      printf("writing regular file '%s' into disk\n", name);
      write_file(&super.s_root, name);
    } else if (S_ISLNK(stat_buf.st_mode)) {   // 在这里为 LNK 类型提供支持，仿照其他分支添加一个 write_symlink 的分支
      printf("writing symlinking file '%s' into disk\n", name);
      write_symlink(&super.s_root, name);
    } else {
      fprintf(stderr, "'%s' has illegal file mode %o\n", name,
              stat_buf.st_mode);
      exit(2);
    }
  }
```

```
}
```

(2) 修改 `main()` 函数，在 `if else` 结构中增加一个分支，调用 `write_symlink()` 函数（下面自行编写），使其不仅支持目录和普通文件的读取，还可以支持符号链接文件的读取。可以使用 `S_ISLNK(stat_buf.stmode)` 判断命令行参数对应的文件是否为符号链接

(3) 修改 `write_directory()` 函数，在 `if else` 结构中增加一个分支，调用你编写的 `write_symlink()` 函数，使其不仅支持目录和普通文件的读取，还可以支持符号链接文件的读取。结构体 `dirent` 的成员变量 `d_type` 可能会用到以下取值：`DT_DIR`（目录）、`DT_LNK`（符号链接）、`DT_REG`（普通文件）

```
void write_directory(struct File *dirf, char *path) {
        DIR *dir = opendir(path);
        if (dir == NULL) {
                perror("opendir");
                return;
        }
        struct File *pdir = create_file(dirf);
        strncpy(pdir->f_name, basename(path), MAXNAMELEN - 1);
        if (pdir->f_name[MAXNAMELEN - 1] != 0) {
                fprintf(stderr, "file name is too long: %s\n", path);
                // File already created, no way back from here.
                exit(1);
        }
        pdir->f_type = FTYPE_DIR;
        for (struct dirent *e; (e = readdir(dir)) != NULL;) {
                if (strcmp(e->d_name, ".") != 0 && strcmp(e->d_name, "..")
!= 0) {
                        char *buf = malloc(strlen(path) + strlen(e->d_name)
+ 2);
                        sprintf(buf, "%s/%s", path, e->d_name);
                        if (e->d_type == DT_DIR) {
                                write_directory(pdir, buf);
                        } else if {
                                write_symlink(pdir, buf); // 在这里添加一个分
支，用来处理链接文件
                        } else {
                                write_file(pdir, buf);
                        }
                        free(buf);
                }
        }
        closedir(dir);
}
```

(4) 可以仿照 `write_file()` 函数编写 `write_symlink()` 函数，实现向磁盘镜像写入符号链接文件的功能，你可以调用 Linux 库函数 `int readlink(char *pathname, char *buf, int bufsiz)` 来读取一个链接文件指向的**目标路径**（这个函数将路径 `pathname` 处的符号链接指向的**目标路径**写入 `buf` 指向的缓冲区，最多写入 `bufsiz` 个字节，返回值是写入的字节数量。详细说明可以在开发机中使用 `man 2 readlink` 命令查阅）一种可能的实现框架如下所示：

```
// 入参含义 : dirf 表示文件创建在哪个目录下，path 是文件在操作系统中的绝对路径
void write_symlink(struct File *dirf, const char *path) {
  int iblk = 0, r = 0, n = sizeof(disk[0].data);
  // 调用 create_file，在 dirf 目录下创建一个新文件 target
```

```
  struct File *target = create_file(dirf);
  char targetpath[2048] = {0};
  // 使用 readlink() 读取符号链接指向的目标路径，path 是符号链接文件所在的路径，targetpath 是符号链接文件的内容(也就是他指向的那个文件的路径)，返回值是 targetpath 的长度 len
  int len = readlink(path, targetpath, 2047);
  // 把目标路径写入文件系统的下一块数据块，手动添加结束符
  memcpy(disk[nextbno].data, targetpath, len);
  disk[nextbno].data[len]='\0';
  // 提取文件名，比如 /tmp/f 会提取出 f
  const char *fname = strrchr(path, '/');
  // 判断是不是纯文件
  if (fname) {
    fname++;
  } else {
    fname = path;
  }
  // 设置文件元信息，也就是文件名，文件大小，文件类型
  strcpy(target->f_name, fname);
  target->f_size = 2048;
  target->f_type = FTYPE_LNK;
  // 建立文件到数据块的映射，注意这里只是逻辑上的映射，也就是先指定哪个磁盘块用来存哪个文件块，并没有真正完成文件内容的加载
  // 这个函数的入参分别表示要映射的文件 target，起始的文件块号 0，要映射到的起始磁盘块号 next_block(BLOCK_DATA)
  save_block_link(target, 0 , next_block(BLOCK_DATA));
}
```

修改文件系统的实现，使其满足：用户程序使用 `open()` 函数打开一个符号链接文件的时候，实际上打开的是其**最终**指向的文件，返回最终指向的文件的文件描述符。被打开的文件可以正常地读写，就像直接打开了最终指向的文件一样，也就是说符号链接文件的 data 段中其实存储的是他所链接的那个文件的路径，相当于需要把这个真实路径解析出来，重入一遍 open 函数即可

```
int open(const char *path, int mode) {
  //......略

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