Lab5

Thinking

Thinking 5.1

> 背景：对于外部设备驱动，使用`kseg1`段，原因是：
>
> - 对`kseg1`段地址的读写，不经过`MMU`映射，不使用高速缓存；
> - 对`kseg0`段地址的读写，不经过`MMU`映射，会使用高速缓存。

> 如果通过`kseg0`读写设备，则对设备的写入会缓存至`Cache`，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。

- 缓存引起的问题：
  - **数据一致性**：如果写入数据被缓存，而缓存尚未被刷新到物理设备；随后CPU或其他设备试图读取该数据时，会读到旧的数据。
  - **写操作的丢失**：如果系统崩溃或重启，而缓存中的数据尚未被写回物理设备，则写操作会丢失。
  - **性能降低**：CPU需要等待缓存一致性操作完成才能继续执行。
- 串口设备和IDE磁盘的差异：
  - **串口设备**（如UART）：
    - **按字节传输**：对数据的一致性要求高(如果写入操作被缓存并乱序执行，那么接收端可能会接收到混乱的数据)
    - **实时性**：串口通信通常需要实时性，延迟的写入操作可能会导致数据丢失或超时。
    - 串口通信的速率通常远低于CPU的速率，因此缓存不会带来明显性能提升。
  - **IDE磁盘**（或其他块存储设备）：
    - **按块传输**：块设备的写入操作通常可以容忍一定程度的乱序。
    - **缓存管理**：`IDE`磁盘本身有缓存机制，用于优化读写性能。如果`CPU`的`Cache`和磁盘的缓存之间没有正确协调，可能会导致数据不一致等问题。
- 解决方案：

- **禁用缓存**：对于直接访问设备的内存区域，应该禁用CPU的Cache。
- **使用标准的I/O接口**：避免直接访问设备的物理地址，而是使用操作系统提供的标准I/O接口。这些接口通常已经处理了缓存等问题。
- **同步和刷新**：在写入设备后，确保执行同步或刷新操作，以确保数据被立即写入设备并清除`CPU`的`Cache`。

Thinking 5.2

- 一个磁盘块中有`FILE2BLK`=`16`个文件控制块；
- 一个目录最多指向`1024`个磁盘块，因此一个目录下最多有`1024*16=16384`个字文件；
- 文件系统中，文件控制块只使用了一级间接指针域，因此文件系统支持的单个文件最大为`4KB*1024=4MB`.

扇区

> 扇区：物理概念，磁盘读写的基本单位
> 扇区读写函数：
> void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs);
> void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);

```
//扇区大小：512字节
#define SECT_SIZE 512
```

磁盘块

> 磁盘块：虚拟概念，操作系统与磁盘交互的最小单位（相邻扇区组合而成）

```
//页面大小：4KB
#define PAGE_SIZE 4096
//磁盘块大小：4KB
#define BLOCK_SIZE PAGE_SIZE
//磁盘块位数：4096*8
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)
//磁盘块数：1024
#define NBLOCK 1024
//磁盘块中的扇区数：8
#define SECT2BLK (BLOCK_SIZE / SECT_SIZE)

struct Block {
    uint8_t data[BLOCK_SIZE];
```

```
    uint32_t type;
} disk[NBLOCK];
```

位图

> 用一个二进制位bit标识磁盘块的使用情况：1表示空闲，0表示占用

```
//存储位图所需块数nbitblock：1[(NBLOCK/BLOCK_SIZE_BIT)向上取整]
nbitblock = (NBLOCK + BLOCK_SIZE_BIT - 1) / BLOCK_SIZE_BIT;
for (i = 0; i < nbitblock; ++i) {
    disk[2 + i].type = BLOCK_BMAP;
}
//所有位图块的每一位置1，设为空闲状态
for (i = 0; i < nbitblock; ++i) {
    memset(disk[2 + i].data, 0xff, BLOCK_SIZE);
}
//若位图有剩余，剩余部分置0（对应磁盘块不存在）
if (NBLOCK != nbitblock * BLOCK_SIZE_BIT) {
    diff = NBLOCK % BLOCK_SIZE_BIT / 8;
    memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BLOCK_SIZE - diff);
}
```

文件控制块

> 普通文件：指向的磁盘块，存储文件内容；
> 目录文件：指向的磁盘块，存储该目录下各个文件对应的文件控制块

```
//一个磁盘块中的文件控制块数：16=4096/256
#define FILE2BLK (BLOCK_SIZE / sizeof(struct File))

#define MAXNAMELEN 128
#define NDIRECT 10
struct File {
        char f_name[MAXNAMELEN]; //文件名
        uint32_t f_size;         //文件大小(以字节为单位)
        uint32_t f_type;         //文件类型:普通文件(FTYPE_REG)和目录(FTYPE_DIR)
        //文件的直接指针：每个文件控制块有10个直接指针，记录：文件的数据块在磁盘上的位置
        uint32_t f_direct[NDIRECT];
        uint32_t f_indirect;    //指向一个间接磁盘块：存储指向文件内容的磁盘块的指针

        struct File *f_dir; //指向文件所属目录
        char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)]; //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）
} __attribute__((aligned(4), packed));
```

Thinking 5.3

在满足磁盘块缓存设计的前提下，使用的内核支持的最大磁盘大小为：`1GB`.

Thinking 5.4

`include/io.h`

```
//读入物理地址paddr处的1字节(8位)整数
//例子: uint8_t data = ioread8(pa);
static inline uint8_t ioread8(u_long paddr) {
    //将内核虚拟地址paddr,转为:kseg1段的内核虚拟地址(paddr+0xA0000000)
    return *(volatile uint8_t *)(paddr | KSEG1);
}
static inline uint16_t ioread16(u_long paddr) {
    return *(volatile uint16_t *)(paddr | KSEG1);
}
static inline uint32_t ioread32(u_long paddr) {
    return *(volatile uint32_t *)(paddr | KSEG1);
}
static inline void iowrite8(uint8_t data, u_long paddr) {
    *(volatile uint8_t *)(paddr | KSEG1) = data;
}
static inline void iowrite16(uint16_t data, u_long paddr) {
    *(volatile uint16_t *)(paddr | KSEG1) = data;
}
static inline void iowrite32(uint32_t data, u_long paddr) {
    *(volatile uint32_t *)(paddr | KSEG1) = data;
}
```

`include/malta.h`

```
#define MALTA_PCIIO_BASE 0x18000000
#define MALTA_IDE_BASE (MALTA_PCIIO_BASE + 0x01f0)
#define MALTA_IDE_DATA (MALTA_IDE_BASE + 0x00)
#define MALTA_IDE_ERR (MALTA_IDE_BASE + 0x01)
#define MALTA_IDE_NSECT (MALTA_IDE_BASE + 0x02)
#define MALTA_IDE_LBAL (MALTA_IDE_BASE + 0x03)
#define MALTA_IDE_LBAM (MALTA_IDE_BASE + 0x04)
#define MALTA_IDE_LBAH (MALTA_IDE_BASE + 0x05)
#define MALTA_IDE_DEVICE (MALTA_IDE_BASE + 0x06)
#define MALTA_IDE_STATUS (MALTA_IDE_BASE + 0x07)
```

```
#define MALTA_IDE_LBA 0xE0
#define MALTA_IDE_BUSY 0x80
#define MALTA_IDE_CMD_PIO_READ 0x20  /* Read sectors with retry */
#define MALTA_IDE_CMD_PIO_WRITE 0x30 /* write sectors with retry */
```

user/include/fs.h

```
//磁盘块大小：4096
#define BLOCK_SIZE PAGE_SIZE
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)
//文件名最大长度
#define MAXNAMELEN 128
//路径最大长度
#define MAXPATHLEN 1024
//文件控制块数：16=4096/256
#define FILE2BLK (BLOCK_SIZE / sizeof(struct File))

struct Super {
        uint32_t s_magic;   // 魔数：一个常量，标识该文件系统
        uint32_t s_nblocks; // 记录本文件系统有多少个磁盘块（这里为1024）
        struct File s_root; // 根目录：其 f_type 为 FTYPE_DIR，f_name 为“/”
};
```

user/include/fd.h

```
#define MAXFD 32
#define FILEBASE 0x60000000
#define FDTABLE (FILEBASE - PDMAP)

//#define PDMAP (4 * 1024 * 1024)   //一个页目录项可映射的地址空间大小:4GB
//#define PTMAP PAGE_SIZE           //页面大小(4096个字节)

//文件表示符i
#define INDEX2FD(i) (FDTABLE + (i)*PTMAP)
//文件表示符i，对应的文件基地址
#define INDEX2DATA(i) (FILEBASE + (i)*PDMAP)

// 文件描述符
struct Fd {
        u_int fd_dev_id;
        u_int fd_offset;
        u_int fd_omode;
};
struct Filefd {
```

```
        struct Fd f_fd;
        u_int f_fileid;
        struct File f_file;
};
```

fs/serv.h

> 块缓存：[DISKMAP,DISKMAP+DISKMAX)作为缓冲区

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

> 脏位：文件系统中，标记该磁盘块缓存是否被修改

```
#define PTE_DIRTY 0x0004
```

user/include/fsreq.h

```
//请求：打开文件
struct Fsreq_open {
        char req_path[MAXPATHLEN];  //request路径
        u_int req_omode;            //request模式
};
//请求：映射
struct Fsreq_map {
        int req_fileid;         //文件ID
        u_int req_offset;       //文件中目标块的偏移
};
//请求：移除文件
struct Fsreq_remove {
        char req_path[MAXPATHLEN];
};
```

Thinking 5.5

文件描述符和定位指针均在用户空间实现，所以`fork`前后的父子进程会共享这些文件相关结构体。

Thinking 5.6

- struct **File**：文件控制块

```
#define MAXNAMELEN 128
#define NDIRECT 10
struct File {
        char f_name[MAXNAMELEN]; //文件名
        uint32_t f_size;         //文件大小(以字节为单位)
        uint32_t f_type;         //文件类型:普通文件(FTYPE_REG)和目录(FTYPE_DIR)
        //文件的直接指针：每个文件控制块有10个直接指针，记录：文件的数据块在磁盘上的位置
        uint32_t f_direct[NDIRECT];
        uint32_t f_indirect;    //指向一个间接磁盘块：存储指向文件内容的磁盘块的指针

        struct File *f_dir; //指向文件所属目录
        char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)]; //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）
} __attribute__((aligned(4), packed));
```

- struct **Fd**：文件描述符

```
// 文件描述符
struct Fd {
        u_int fd_dev_id;    //外设id
        u_int fd_offset;    //读写的当前位置（偏移量）
        u_int fd_omode;     //文件打开方式，如只读，只写等
};
```

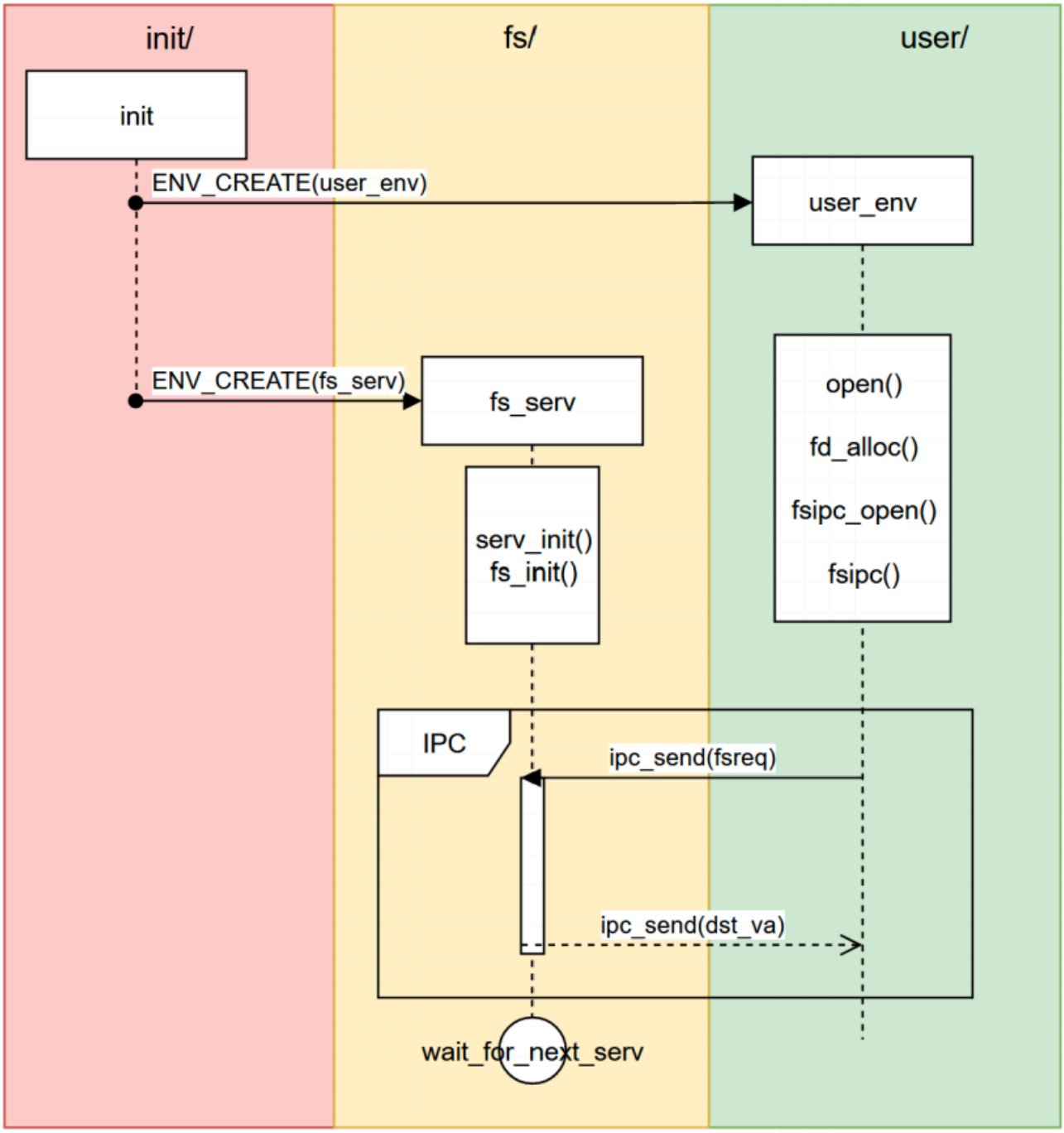
- struct **Filefd**：记录文件详细信息

```
struct Filefd {
        struct Fd f_fd;     //文件描述符
        u_int f_fileid;     //文件id
        struct File f_file; //对应的文件控制块
};
```

Thinking 5.7

文件系统服务:

- 同步消息：（黑三角箭头+黑实线）消息的发送者将进程控制传递给消息的接收者，然后**暂停活动**，**等待**息接收者的回应消息。
- 返回消息：（开三角箭头+虚线）返回消息和同步消息结合使用，因为**异步消息**不进行等待，所以不需要知道返回值。



Exercise

Exercise 5.1

> 背景：要编写用户空间的磁盘驱动程序，对设备的读写通过**系统调用**实现。
> `sys_write_dev`，`sys_read_dev`函数：以用户虚拟地址、设备的物理地址和读写的长度（按字节计数）、重复次数（重复读或写同一个物理地址次数），作为参数，在内核空间中完成

> I/O 操作。
> 前提条件：读写的长度len只能是1/2/4个字节
> 要点：获取物理地址`src_pa`,`dst_pa`,以通过`ioread8`,`iowrite8`函数读写。
>
> 1. 求`src_pa`:将用户虚拟地址，转为物理地址(`MMU`映射)
> 2. 求`dst_pa`:传入的参数`pa`

sys_write_dev函数

```
//将用户虚拟地址va处,长度为len的数据,写入设备物理地址pa处
int sys_write_dev(u_int va, u_int pa, u_int len) {
    //1.验证地址段[va,va+len)的合法性:
    if(is_illegal_va_range(va,len)) return -E_INVAL;
    if(len!=1 && len!=2 && len!=4) return -E_INVAL;

    //2.验证设备物理地址pa的合法性:
    /*实验中允许的范围为:
    console: [0x180003F8, 0x18000418), disk: [0x180001F0, 0x180001F8)*/
    if(!(0x180003f8<=pa && pa+len<=0x18000418) && !(0x180001f0<=pa &&
pa+len<=0x180001f8)){
        return -E_INVAL;
    }

    //3.获取用户虚拟地址va对应的页控制块p
    struct Page *pp;
    if((pp=page_lookup(cur_pgdir,va,NULL))==NULL) return -E_INVAL;
    u_long src_pa=page2pa(pp)+(va&0xfff);

    //4.将[va,va+len)处的数据，复制至[pa,pa+len)处
    if(len==1){
        uint8_t data=ioread8(src_pa);
        iowrite8(data,dst_pa);
    }else if(len==2){
        uint16_t data=ioread16(src_pa);
        iowrite16(data,dst_pa);
    }else{
        uint32_t data=ioread32(src_pa);
        iowrite32(data,dst_pa);
    }
    return 0;
}
```

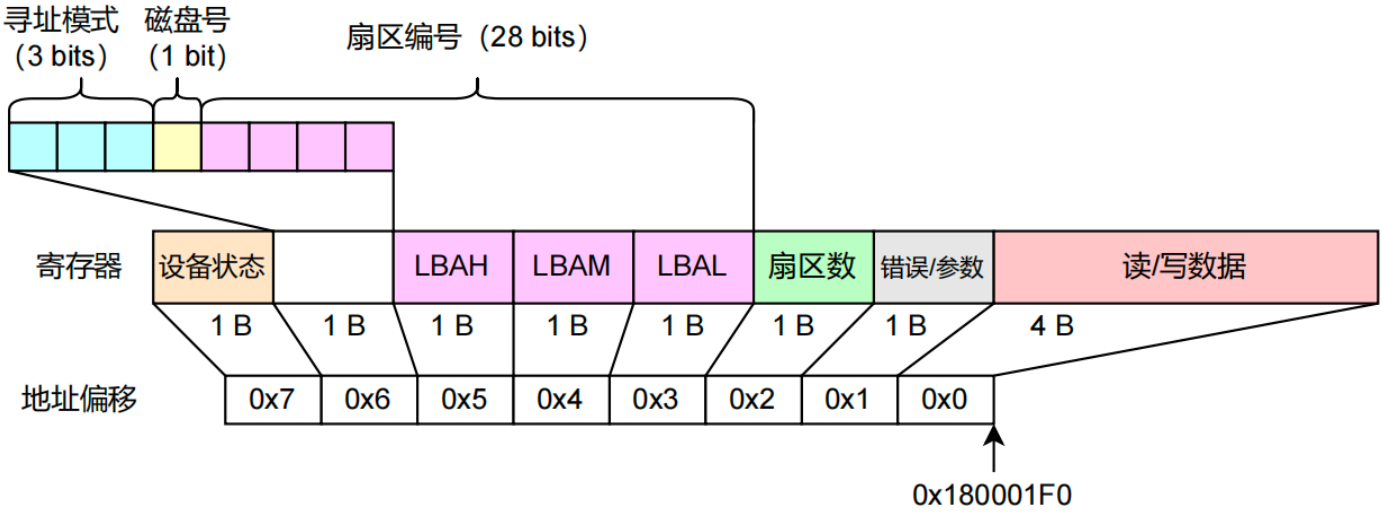
sys_read_dev函数

> 要点：类`sys_write_dev`函数，交换`src_pa`和`dst_pa`.
> 将设备物理地址pa处,长度为len的数据,读至用户进程虚拟地址[va,va+len)处

Notes

磁盘寻址：如何定位扇区？LBA模式（逻辑块寻址）

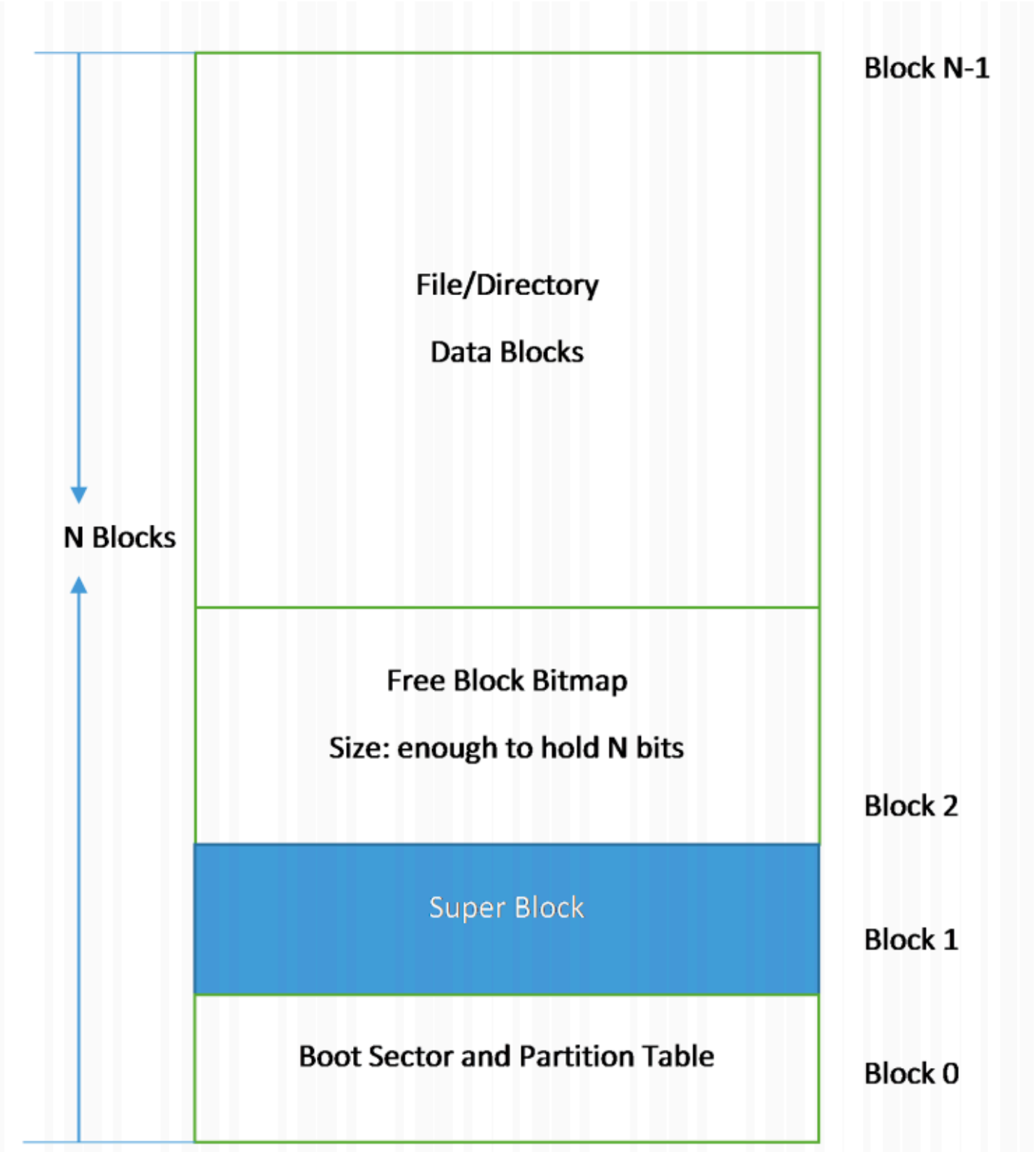
> 将磁盘视作一个线性的字节序列，每个扇区都有唯一的编号。
> 扇区编号有28位，最多可寻址2^28个扇区，即：128GB的磁盘空间。



磁盘文件系统

磁盘块

> **磁盘块**是**操作系统与磁盘交互**的**最小单位**。
> 操作系统将相邻的扇区组合在一起，形成磁盘块进行整体操作，减小了因扇区过多带来的寻址困难；
> 磁盘块是虚拟概念，大小由操作系统决定，一般为`2^n`个扇区。
> （扇区是物理概念，是磁盘读写的基本单位）
>
> `Block 0`(4096个字节):引导扇区，分区表
> `Block 1`:超级块，定义为：`Super`结构体，描述文件系统的基本信息




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

超级块

```
struct Super {
        uint32_t s_magic;   // 魔数：一个常量，标识该文件系统
        uint32_t s_nblocks; // 记录本文件系统有多少个磁盘块（这里为1024）
        struct File s_root; // 根目录：其 f_type 为 FTYPE_DIR，f_name 为“/”
};
```

磁盘空闲块机制：位图

- 二进制位`bit`标识磁盘中的一个磁盘块的使用情况（1 表示空闲，0 表示占用）。

文件控制块:`File`结构体

> 注：每个磁盘块大小为`4KB`，十个直接指针能表示最大`40KB`的文件。
> 文件大于`40KB`时，使用间接指针。

```
#define MAXNAMELEN 128
#define NDIRECT 10
struct File {
        char f_name[MAXNAMELEN]; //文件名
        uint32_t f_size;         //文件大小(以字节为单位)
        uint32_t f_type;         //文件类型:普通文件(FTYPE_REG)和目录(FTYPE_DIR)
        //文件的直接指针：每个文件控制块有10个直接指针，记录：文件的数据块在磁盘上的位置
        uint32_t f_direct[NDIRECT];
        uint32_t f_indirect;    //指向一个间接磁盘块：存储指向文件内容的磁盘块的指针

        struct File *f_dir; //指向文件所属目录
        char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void
*)]; //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）
} __attribute__((aligned(4), packed));
```

- 普通文件：指向的磁盘块存储着文件内容
- 目录文件：指向的磁盘块存储着该目录下个文件对应的文件控制块

> 查找文件：从超级块中读取根目录的文件控制块；挨个查看当前目录所包含的文件是否与下一级目标文件同名。

块缓存

使用虚拟内存实现磁盘块缓存。
文件系统服务是一个用户进程。

一个进程可拥有：`4GB`的虚拟内存空间，将`[DISKMAP, DISKMAP+DISKMAX)`(即：`[0x1000 0000, 0x4fff ffff]`作为缓冲区，当磁盘读入内存时，用于映射相关的页。

文件描述符

> 存储文件的基本信息，用户进程中关于文件的状态。

函数

`fs/ide.c`

wait_ide_ready函数

> 检查`IDE`是否就绪:

```
static uint8_t wait_ide_ready() {
        uint8_t flag;
        while (1) {
       //传入flag的虚拟地址&va:
       //将IDE物理地址MALTA_IDE_STATUS处,长度为1字节的数据,拷贝至flag处
                panic_on(syscall_read_dev(&flag, MALTA_IDE_STATUS, 1));
               if ((flag & MALTA_IDE_BUSY) == 0) break;
               syscall_yield();
        }
        return flag;
}
```

ide_read函数

> 将IDE磁盘扇区的数据，读至设备缓冲区中:
> 读取diskno号的磁盘上,[secno,secno+nsecs)号扇区到dst指定地址
> 扇区大小：512个字节

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs) {
    uint8_t temp;
    u_int offset = 0, max = nsecs + secno;
    panic_on(diskno >= 2);

    //轮流读取扇区
    while (secno < max) {
        temp = wait_ide_ready();
        // Step 1:操作扇区数目写入NSECT寄存器
```

```
        temp = 1;
        /*将用户虚拟地址[va,va+len)处的数据，写入物理地址pa处
        sys_write_dev(u_int va, u_int pa, u_int len);
        #define MALTA_IDE_NSECT 0x180001f2*/
        panic_on(syscall_write_dev(&temp, MALTA_IDE_NSECT, 1));

        // Step 2: 目标扇区号的[7:0]位写入LBAL寄存器
        temp = secno & 0xff;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAL, 1));

        // Step 3: 目标扇区号的[15:8]位写入LBAM寄存器
        temp = (secno>>8) & 0xff;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAM, 1));

        // Step 4: 目标扇区号的[23:16]位写入LBAH寄存器
        temp = (secno>>16) & 0xff;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAH, 1));

        // Step 5: 设置操作扇区号的 [27:24] 位；设置扇区寻址模式；设置磁盘编号
        temp = ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
        panic_on(syscall_write_dev(&temp, MALTA_IDE_DEVICE, 1));

        // Step 6: 设置IDE设备为读状态
        temp = MALTA_IDE_CMD_PIO_READ;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_STATUS, 1));

        // Step 7: 等待IDE设备就绪
        temp = wait_ide_ready();

        // Step 8: 读取该扇区的数据:
        //物理地址MALTA_IDE_DATA处的数据，读至目标虚拟地址dst + offset + i * 4处
        for (int i = 0; i < SECT_SIZE / 4; i++) {
            panic_on(syscall_read_dev(dst + offset + i * 4, MALTA_IDE_DATA, 4));
        }

        // Step 9: 检查IDE设备状态
        panic_on(syscall_read_dev(&temp, MALTA_IDE_STATUS, 1));

        offset += SECT_SIZE;
        secno += 1;
    }
}
```

ide_write函数

> 将设备缓冲区中数据，写入IDE磁盘中的指定扇区:

> 将指定地址src处的数据,写入diskno号的磁盘上[secno,secno+nsecs)号扇区

```
void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs) {
    uint8_t temp;
    u_int offset = 0, max = nsecs + secno;
    panic_on(diskno >= 2);

    // Write the sector in turn
    while (secno < max) {
        temp = wait_ide_ready();
        // Step 1: Write the number of operating sectors to NSECT register
      temp = 1;
      panic_on(syscall_write_dev(&temp, MALTA_IDE_NSECT, 1));

        // Step 2: Write the 7:0 bits of sector number to LBAL register
      temp = secno & 0xff;
      panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAL, 1));

        // Step 3: Write the 15:8 bits of sector number to LBAM register
        temp = (secno>>8) & 0xff;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAM, 1));

        // Step 4: Write the 23:16 bits of sector number to LBAH register
        temp = (secno>>16) & 0xff;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_LBAH, 1));

        // Step 5: Write the 27:24 bits of sector number, addressing mode
        // and diskno to DEVICE register
        temp = ((secno >> 24) & 0x0f) | MALTA_IDE_LBA | (diskno << 4);
        panic_on(syscall_write_dev(&temp, MALTA_IDE_DEVICE, 1));

        // Step 6: Write the working mode to STATUS register
        temp = MALTA_IDE_CMD_PIO_WRITE;
        panic_on(syscall_write_dev(&temp, MALTA_IDE_STATUS, 1));

        // Step 7: Wait until the IDE is ready
        temp = wait_ide_ready();

        // Step 8: Write the data to device
        //int sys_write_dev(u_int va, u_int pa, u_int len);
        //用户虚拟地址(dst + offset + i * 4)处的数据，写入物理地址MALTA_IDE_DATA处
        for (int i = 0; i < SECT_SIZE / 4; i++) {
          panic_on(syscall_write_dev(dst + offset + i * 4, MALTA_IDE_DATA, 4));
        }

        // Step 9: Check IDE status
        panic_on(syscall_read_dev(&temp, MALTA_IDE_STATUS, 1));
```

```
        offset += SECT_SIZE;
        secno += 1;
    }
}
```

`tools/fsformat.c`

变量/常量

```
#define PAGE_SIZE 4096
#define NBLOCK 1024 //磁盘中的Block数
uint32_t nbitblock; // the number of bitmap blocks.
uint32_t nextbno;   // next availiable block.
enum {      //Block类型
    BLOCK_FREE = 0,
    BLOCK_BOOT = 1,
    BLOCK_BMAP = 2,
    BLOCK_SUPER = 3,
    BLOCK_DATA = 4,
    BLOCK_FILE = 5,
    BLOCK_INDEX = 6,
};
struct Block {
    uint8_t data[BLOCK_SIZE];
    uint32_t type;
} disk[NBLOCK];
```

reverse函数

> 大小端转换

```
void reverse(uint32_t *p) {
    uint8_t *x = (uint8_t *)p;
    uint32_t y = *(uint32_t *)x;
    x[3] = y & 0xFF;
    x[2] = (y >> 8) & 0xFF;
    x[1] = (y >> 16) & 0xFF;
    x[0] = (y >> 24) & 0xFF;
}
```

reverse_block函数

> 接收一个指向`Block`结构体的指针`b`作为参数，并根据`Block`的类型来反转其中的特定字段

```
void reverse_block(struct Block *b) {
    int i, j;
    struct Super *s;
    struct File *f, *ff;
    uint32_t *u;

    switch (b->type) {
    //BLOCK_FREE和BLOCK_BOOT:不进行任何操作
    case BLOCK_FREE:
    case BLOCK_BOOT:
        break;
    //BLOCK_SUPER:
    //1.反转超级块中的 s_magic 和 s_nblocks 字段;
    //2.反转根目录文件 (s_root) 的 f_size、f_type、f_direct 数组和 f_indirect 字段
    case BLOCK_SUPER:
        s = (struct Super *)b->data;
        reverse(&s->s_magic);
        reverse(&s->s_nblocks);

        ff = &s->s_root;
        reverse(&ff->f_size);
        reverse(&ff->f_type);
        for (i = 0; i < NDIRECT; ++i) {
            reverse(&ff->f_direct[i]);
        }
        reverse(&ff->f_indirect);
        break;
    //BLOCK_FILE:
    //1.遍历 Block 中的所有 File 结构体 (直到找到一个文件名为空的 File 为止)
    //2.对于每个有效的 File 结构体，反转其 f_size、f_type、f_direct 数组和 f_indirect 字
段
    case BLOCK_FILE:
        f = (struct File *)b->data;
        for (i = 0; i < FILE2BLK; ++i) {
            ff = f + i;
            if (ff->f_name[0] == 0) {
                break;
            } else {
                reverse(&ff->f_size);
                reverse(&ff->f_type);
                for (j = 0; j < NDIRECT; ++j) {
                    reverse(&ff->f_direct[j]);
                }
                reverse(&ff->f_indirect);
            }
```

```
        }
        break;
    //BLOCK_INDEX 和 BLOCK_BMAP:
    //这两个类型的块通常包含一系列 uint32_t 类型的值：遍历整个块并反转每个 uint32_t 类型的值。
    case BLOCK_INDEX:
    case BLOCK_BMAP:
        u = (uint32_t *)b->data;
        for (i = 0; i < BLOCK_SIZE / 4; ++i) {
            reverse(u + i);
        }
        break;
    }
}
```

init_disk函数

> 初始化：将所有Block标为空闲块

```
#define NBLOCK 1024     //磁盘块数
//#define BLOCK_SIZE PAGE_SIZE
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)
void init_disk() {
    int i, diff;

    // Step 1: Mark boot sector block.
    disk[0].type = BLOCK_BOOT;

    // Step 2: Initialize boundary.
    //nbitblock为所需位图块数量(bitblock)
    nbitblock = (NBLOCK + BLOCK_SIZE_BIT - 1) / BLOCK_SIZE_BIT;
    nextbno = 2 + nbitblock;

    // Step 2: Initialize bitmap blocks.
    for (i = 0; i < nbitblock; ++i) {
        disk[2 + i].type = BLOCK_BMAP;
    }
    for (i = 0; i < nbitblock; ++i) {
       //将memset 将位图中的每一个字节 (Byte) 都设成 0xff，即将所有位图块的每一位都设为 1，表示磁盘块处于空闲状态
        memset(disk[2 + i].data, 0xff, BLOCK_SIZE);
    }
       //如果位图有剩余，将最后一块位图块中不存在的部分设为 0
    if (NBLOCK != nbitblock * BLOCK_SIZE_BIT) {
        diff = NBLOCK % BLOCK_SIZE_BIT / 8;
        memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BLOCK_SIZE - diff);
    }
```

```
    // Step 3: Initialize super block.
    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, "/");
}
```

next_block函数

> 获取下一个block的ID，设置其`type`：

```
int next_block(int type) {
    disk[nextbno].type = type;
    return nextbno++;
}
```

flush_bitmap函数

> 将磁盘块使用情况，存储在bitmap中:

```
void flush_bitmap() {
    int i;
    // update bitmap, mark all bit where corresponding block is used.
    for (i = 0; i < nextbno; ++i) {
        ((uint32_t *)disk[2+i/BLOCK_SIZE_BIT].data)[(i%BLOCK_SIZE_BIT)/32]&=~
(1<<(i%32));          //将第i%32位清除为0
    }
}
```

finish_fs函数

> 将block序列写入文件名为`name`的文件:

```
void finish_fs(char *name) {    //字符串指针name指向要写入数据的目标文件名
    int fd, i;

    //1.将super块的内容，复制至：disk数组第二个元素disk[1]的data字段
    memcpy(disk[1].data, &super, sizeof(super));

    //2.使用open函数打开/创建一个文件，并返回一个文件描述符fd
    //读写模式打开，权限设置为0666
```

```
    fd = open(name, O_RDWR | O_CREAT, 0666);
    //3.对 disk 数组中的每个块调用 reverse_block 函数
    for (i = 0; i < 1024; ++i) {
#ifdef CONFIG_REVERSE_ENDIAN
        reverse_block(disk + i);
#endif
        ssize_t n = write(fd, disk[i].data, BLOCK_SIZE);
        assert(n == BLOCK_SIZE);
    }

    //关闭文件描述符fd
    close(fd);
}
```

save_block_link函数

> 保存文件(`File`结构体)中的一个块(block)链接：
> 将文件的一个逻辑块号（`nblk`）与磁盘上的一个物理块号（`bno`）关联起来

```
void save_block_link(struct File *f, int nblk, int bno) {
    //1.确保逻辑块号 nblk 小于 NINDIRECT:该文件使用的块数量，不超过直接块和间接块的总和
    assert(nblk < NINDIRECT);

    if (nblk < NDIRECT) {       //(1)使用文件的直接块数组 f->f_direct 来保存块链接
        f->f_direct[nblk] = bno;
    } else {                //(2)使用文件的间接块保存块链接
       //如果f->f_indirect=0，则：该文件没有间接块，因此需要创建一个新的间接块
        if (f->f_indirect == 0) {
              f->f_indirect = next_block(BLOCK_INDEX);
        }
        ((uint32_t *)(disk[f->f_indirect].data))[nblk] = bno;
    }
}
```

make_link_block函数

> 为文件`dirf`中逻辑块`nblk`分配一个磁盘物理块，并保存其块链接。

```
int make_link_block(struct File *dirf, int nblk) {
    int bno = next_block(BLOCK_FILE);  //分配新的块，块号为bno
    save_block_link(dirf, nblk, bno);  //在文件dirf中，保存新的块链接
    dirf->f_size += BLOCK_SIZE;        //更新文件dirf的大小
    return bno;         //返回新的磁盘物理块号bno
}
```

`fs/fs.c`

disk_addr函数

> 计算缓存中：磁盘块`blockno`对应的虚存起始地址
> 块缓存机制：使用`[DISKMAP,DISKMAP+DISKMAX)`共`4GB`的虚拟地址空间，作为磁盘块缓冲区。
> 块缓存起始地址：DISKMAP: 0x1000 0000;
> 磁盘块大小：BLOCK_SIZE: 4KB;
> 磁盘块数：NBLOCK: 1024.

```
void *disk_addr(u_int blockno) {
        return (void *)(DISKMAP+BLOCK_SIZE*blockno);
}
```

block_is_mapped函数

> va_is_mapped:检查虚拟地址va是否映射至一个block(检查一级/二级页表项的有效位PTE_V)

```
int va_is_mapped(void *va) {
    return (vpd[PDX(va)] & PTE_V) && (vpt[VPN(va)] & PTE_V);
}
```

> block_is_mapped:检查磁盘块blockno是否映射至cache，返回虚拟地址`va`
> 即检查磁盘块blockno对应的虚存地址va

```
void *block_is_mapped(u_int blockno) {
    //获取磁盘块blockno在cache中的虚拟地址va
    void *va = disk_addr(blockno);
    if (va_is_mapped(va)) return va;
    return NULL;
}
```

block_is_dirty函数

> va_is_dirty:检查虚拟地址`va`是否被修改(检查虚拟页号的脏位`PTE_DIRTY`)

```
int va_is_dirty(void *va) {
    return vpt[VPN(va)] & PTE_DIRTY;
```

```
}
```

> block_is_dirty:检查磁盘块blockno是否dirty
> 即检查blockno对应的虚存地址va

```
int block_is_dirty(u_int blockno) {
        void *va = disk_addr(blockno);
        return va_is_mapped(va) && va_is_dirty(va);
}
```

dirty_block函数

> 将磁盘块`blockno`标记为dirty(缓存页面已更改，需要重写回磁盘)

```
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;
    return syscall_mem_map(0, va, 0, va, PTE_D | PTE_DIRTY);
}
```

write_block函数

> 将当前`blockno`对应的磁盘块内容，写回磁盘

```
void write_block(u_int blockno) {
    //1.检查当前block是否已经映射
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }

    //2.将磁盘块缓存位置的数据，写回IDE磁盘(一个磁盘块上8个扇区)
    void *va = disk_addr(blockno);
    //将指定地址src处的数据,写入diskno号的磁盘上[secno,secno+nsecs)号扇区
    //void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs);
    //将虚拟地址va处的数据，写入0号磁盘上[blockno * SECT2BLK,(blockno+1)*SECT2BLK)号扇区
    ide_write(0, blockno * SECT2BLK, va, SECT2BLK);    //SECT2BLK=8.
}
```

map_block函数

> 将`blockno`对应磁盘块载入内存时，检查该磁盘块是否已经映射至内存;如果没有,分配一页内

> 存

```
int map_block(u_int blockno) {
        // Step 1: If the block is already mapped in cache, return 0.

        /*void *block_is_mapped(u_int blockno)函数:
            检查磁盘块blockno是否映射至cache;获取其在cache中的虚拟地址*/
        void *va=block_is_mapped(blockno);
        if(va!=NULL) return 0;      //已经映射

        // Step 2: Alloc a page in permission 'PTE_D' via syscall.
        // Hint: Use 'disk_addr' for the virtual address.
        /*syscall_mem_alloc函数:
            int syscall_mem_alloc(u_int envid, void *va, u_int perm);
            在envid对应进程的地址空间中,为虚拟地址va处分配一个物理页面*/
        int r=syscall_mem_alloc(0,disk_addr(blockno),PTE_D);
        return r;
}
```

unmap_block函数

> 结束使用`blockno`对应的磁盘块时,释放对应物理内存,解除`va`和物理页面的映射

```
void unmap_block(u_int blockno) {
        //1.获取blockno对应磁盘块映射到的虚拟地址va
        void *va=block_is_mapped(blockno);

        //2.该磁盘块缓存被修改，写回磁盘
        if(!block_is_free(blockno) && block_is_dirty(blockno)){
                write_block(blockno);
        }

        //3.解除va和物理页面的映射
        syscall_mem_unmap(0,va);
        user_assert(!block_is_mapped(blockno));
}
```

file_get_block函数

```
int file_get_block(struct File *f, u_int filebno, void **blk) {
        int r;
        u_int diskbno;
        u_int isnew;
```

```
        // Step 1: find the disk block number is `f` using `file_map_block`.
        if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) return r;

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

`user/lib/fd.c`

dev_lookup函数

```
int dev_lookup(int dev_id, struct Dev **dev) {
    for (int i = 0; devtab[i]; i++) {
        if (devtab[i]->dev_id == dev_id) {
            *dev = devtab[i];
            return 0;
        }
    }

    debugf("[%08x] unknown device type %d\n", env->env_id, dev_id);
    return -E_INVAL;
}
```

fd_lookup函数

> 获取`fdnum`对应的文件描述符,其指针存储在`fd`的地址
> #define INDEX2FD(i) (FDTABLE + (i)*PTMAP)

```
int fd_lookup(int fdnum, struct Fd **fd) {
        u_int va;
        if (fdnum >= MAXFD) {return -E_INVAL;   //超出范围
        va = INDEX2FD(fdnum);       //编号fdnum的文件表示符的虚拟地址
        if ((vpt[va / PTMAP] & PTE_V) != 0) {
                *fd = (struct Fd *)va;
                return 0;
        }
        return -E_INVAL;
}
```

fd_alloc函数

> 寻找未映射的编号最小的文件描述符fd（不进行分配,由其调用者进行分配）

> 描述符表`FDTABLE`:包含`MAXFD=32`个文件描述符

```
//使得fd指向：映射页面的虚拟地址
int fd_alloc(struct Fd **fd) {
        u_int va;
        u_int fdno;

        for (fdno = 0; fdno < MAXFD - 1; fdno++) {
                va = INDEX2FD(fdno);
                if ((vpd[va/PDMAP]&PTE_V)== 0||(vpt[va/PTMAP]&PTE_V)==0) {
                        *fd = (struct Fd *)va;
                        return 0;
                }
        }
        return -E_MAX_OPEN;
}
```

fd2num函数

> 获取文件描述符`fd`的编号

```
int fd2num(struct Fd *fd) {
        return ((u_int)fd - FDTABLE) / PTMAP;
}
```

num2fd函数

> 获取编号对应的文件描述符

```
int num2fd(int fd) {
        return fd * PTMAP + FDTABLE;
}
```

fd2data函数

> 获取文件表示符fd，对应的文件基地址
> #define INDEX2DATA(i) (FILEBASE + (i)*PDMAP)

```
void *fd2data(struct Fd *fd) {
        return (void *)INDEX2DATA(fd2num(fd));
}
```

`user/lib/fsipc.c`

fsipc函数

> 向文件服务器发送IPC请求:
> type:请求码
> fsreq:包含额外请求信息的页面(通常是fsipcbuf)
> dstva:接收回复页面的虚拟地址

```
static int fsipc(u_int type, void *fsreq, void *dstva, u_int *perm) {
        u_int whom;
        // Our file system server must be the 2nd env.

        //sys_ipc_try_send函数:int sys_ipc_try_send(u_int envid, u_int value, u_int
srcva, u_int perm);
        //将值value,发送方进程curenv中srcva对应的页面,发送至envid对应进程.
        ipc_send(envs[1].env_id, type, fsreq, PTE_D);

        //sys_ipc_recv函数:当前进程curenv等待从其他进程接收消息:一个值,一个页面
        return ipc_recv(&whom, dstva, perm);
}
```

fsipc_open函数

```
//向文件服务器发送"打开文件"的请求:
int fsipc_open(const char *path, u_int omode, struct Fd *fd) {
        u_int perm;
        struct Fsreq_open *req;

        req = (struct Fsreq_open *)fsipcbuf;    //新建请求
        // 路径超出最大长度
        if (strlen(path) >= MAXPATHLEN) {
                return -E_BAD_PATH;
        }
        strcpy((char *)req->req_path, path);
        req->req_omode = omode;
        return fsipc(FSREQ_OPEN, req, fd, &perm);
}
```

fsipc_map函数

> 映射请求:发送文件ID(`fileid`)和文件中目标块的偏移(`offset`),返回包含该块的页面的映射
> 调用 `fsipc` 函数： static int fsipc(u_int type, void *fsreq, void *dstva, u_int perm){

```
/.../
ipc_send(envs[1].env_id, type, fsreq, PTE_D);
return ipc_recv(&whom, dstva, perm);
}
```

```
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;

        //与文件服务器的通信:
        //1.将值FSREQ_MAP, req映射的页面, 发送至: 文件服务进程envs[1].env_id;
        //2.从文件服务进程, 接收: 值r, 页面映射至dstva
        if ((r = fsipc(FSREQ_MAP, req, dstva, &perm)) < 0) {
                return r;
        }

        //除了PTE_D(可写位),PTE_LIBRARY(共享位)外, 只有PTE_V(有效位)被设置:
        if ((perm & ~(PTE_D | PTE_LIBRARY)) != (PTE_V)) {
                user_panic("fsipc_map: unexpected permissions %08x for dstva
%08x", perm, dstva);
        }
        return 0;
}
```