注意事项

- 通过页表来判断是否被使用，即`(vpd[PDX(va)] & PTE_V)`与`(vpt[VPN(va)] & PTE_V`
- 注意指针的强制类型转化

实验目的

5.1 实验目的

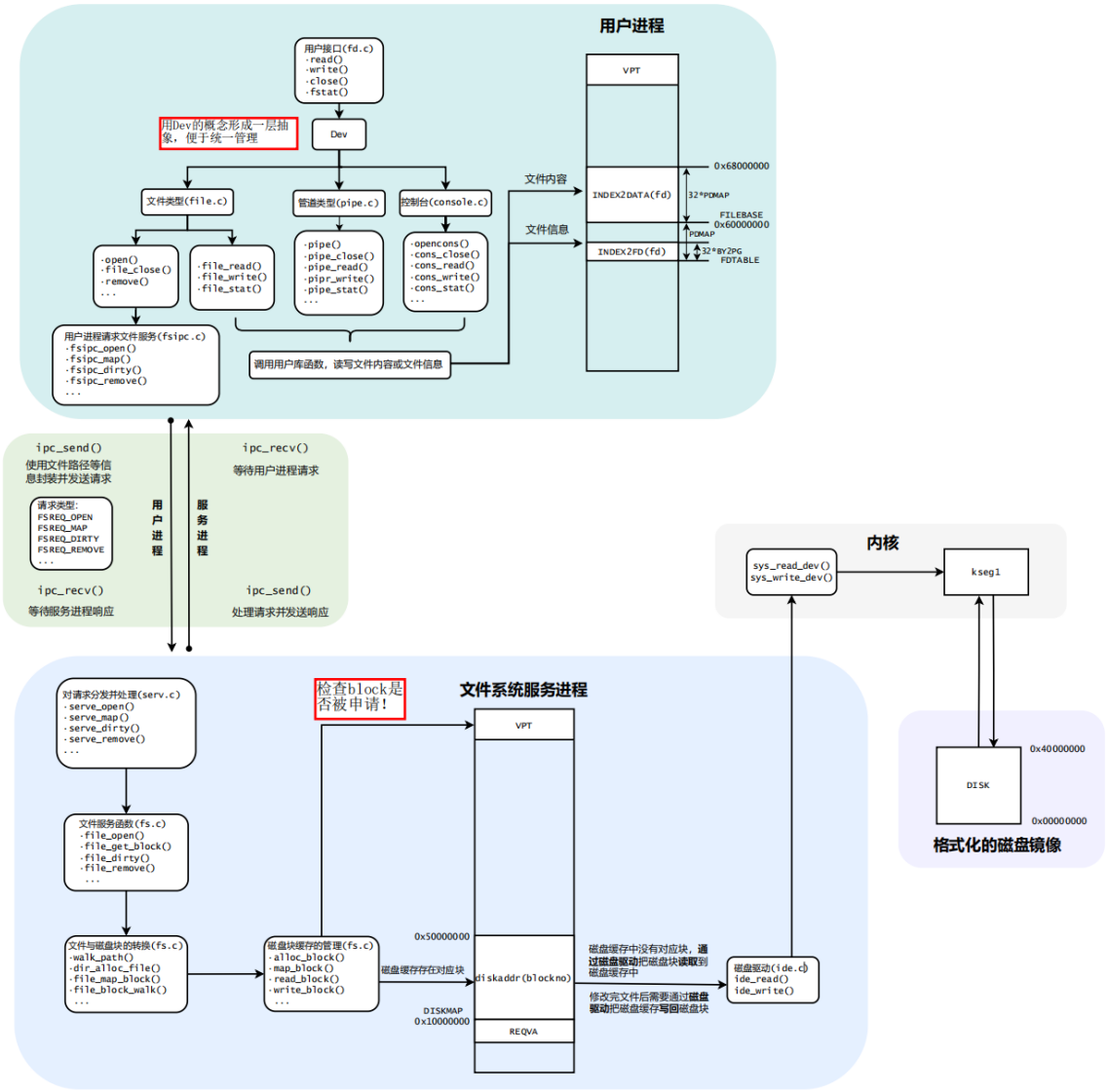
1. 了解文件系统的基本概念和作用。
2. 了解普通磁盘的基本结构和读写方式。
3. 了解实现设备驱动的方法。
4. 掌握并实现文件系统服务的基本操作。
5. 了解微内核的基本设计思想和结构。

在之前的实验中，我们把所有的程序和数据都存放在内存中。然而内存空间的大小是有限的，而且内存中的数据存在易失性问题。因此有些数据必须保存在磁盘、光盘等外部存储设备上，这些外部存储设备能够长期保存大量的数据，且便于将数据装载到不同进程的内存空间进行共享。为了便于管理和访问存放在外部存储设备上的数据，在操作系统中引入了文件系统。在文件系统中，文件是数据存储和访问的基本单位。对于用户而言，文件系统可以屏蔽访问外存数据的复杂性。

在本实验中，我们拟实现一个精简的文件系统，其中需要对三种设备进行统一管理，即文件设备（file，即狭义的“文件”）、控制台（console）和管道（pipe）。其中，后两者将在下一个实验“管道与 Shell”中进行使用

实验总览

- tools目录中存放的是**构建时辅助工具**的代码，fdformat工具——创建磁盘镜像
- fs目录中存放的是**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
    - fs.c: 实现文件系统的基本功能函数
    - ide.c: 通过系统调用与磁盘镜像进行交互
    - serv.c: 进程的主干函数
- user/lib
- 目录下存放了用户程序的库函数：允许用户程序使用统一的接口，抽象地操作磁盘文件系统中的文件，以及控制台和管道等虚拟的文件。
    - fsipc.c: 实现与文件系统服务进程的交互
    - file.c: 实现文件系统的用户接口
    - fd.c: 实现文件描述符



这里我们给出了文件系统结构中部分函数可能的调用参考（图 5.8），希望同学们认真理解每个文件、函数的作用和之间的关系。

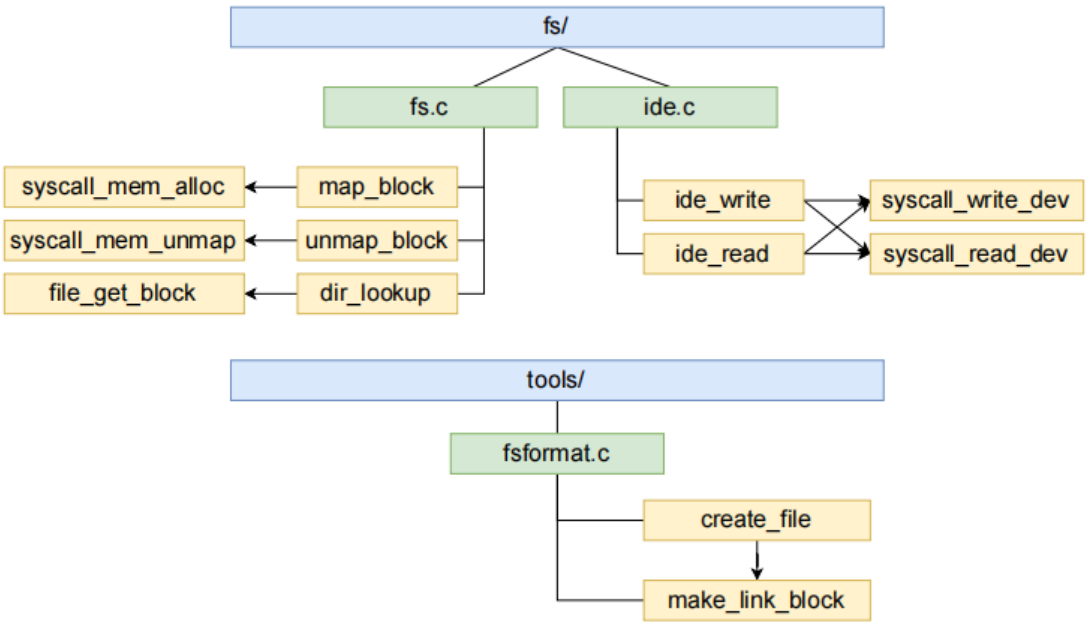


图 5.8: fs/下部分函数调用关系参考

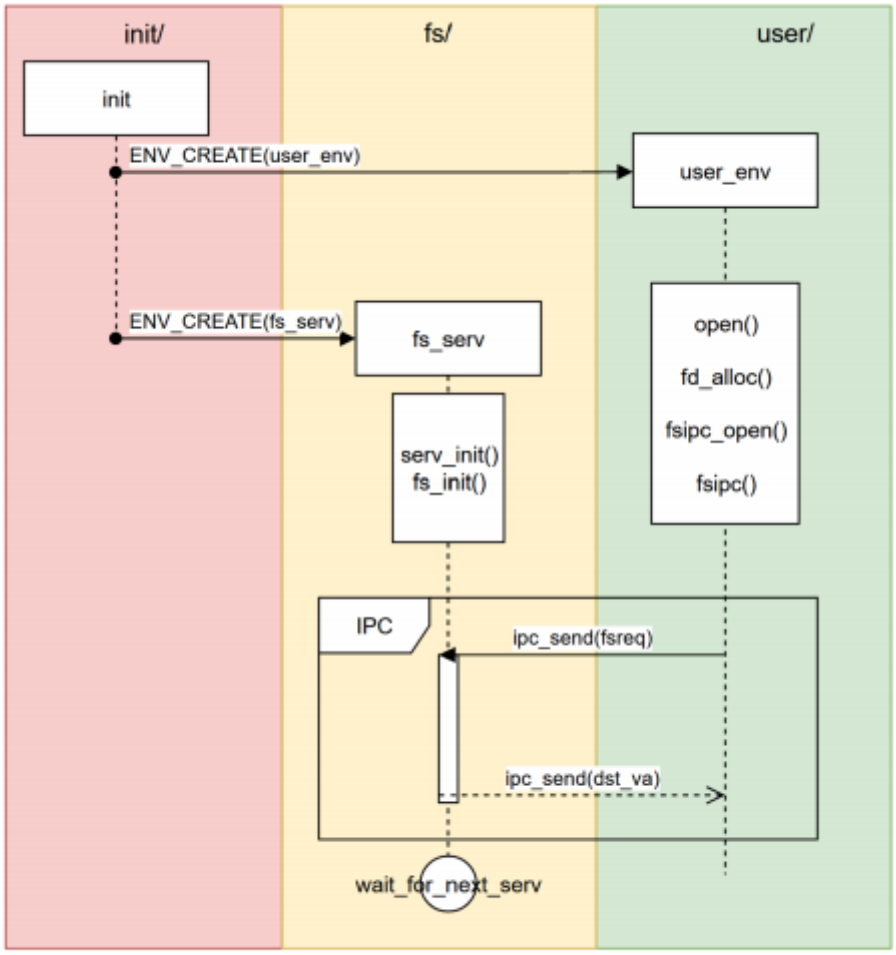


图 5.9: 文件系统服务时序图

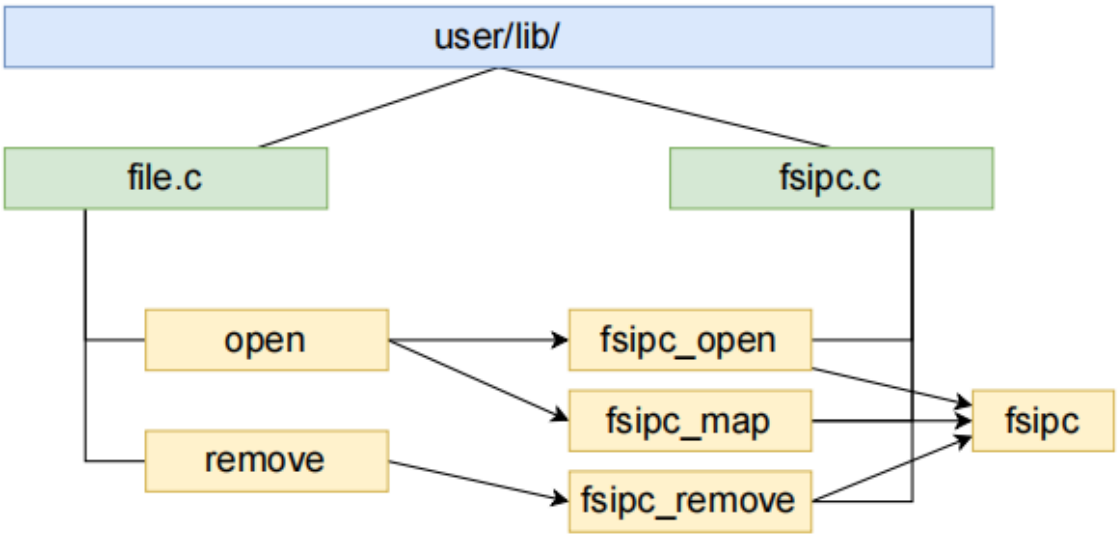


图 5.10: `user/lib` 下部分函数调用关系参考

> 啊，操作系统真精巧啊，小小zyt一辈子都理解不了其中全部奥妙！！！

常用工具

```
// 判断 block 是否空闲，1为空闲，0为非空闲
bitmap[blockno / 32] & (1 << (blockno % 32));
```

```
// 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
(vpd[PDX(va)] & PTE_V)
(vpt[VPN(va)] & PTE_V);
```

```
// 遍历目录下的所有文件
nblock = dir->f_size / BLOCK_SIZE;
// 遍历目录的所有磁盘块
for (int i = 0; i < nblock; i++) {
    void *blk;
    try(file_get_block(dir, i, &blk)); // 获取文件中第i个磁盘块
    struct File *files = (struct File *)blk;
    // 遍历每个磁盘块下的所有File结构体
    for (struct File *f = files; f < files + FILE2BLK; ++f) {
        /* do something
            such as:
            if (strcmp(name, f->f_name) == 0) {
                *file = f;
                f->f_dir = dir;
                return 0;
            }
            */
    }
}
```

```
// 将一个 struct Fd * 型的指针转换为 struct Filefd * 型的指针
// 实际上就是用户进程申请了一个Fd，然后Fd本身是一个指针，在请求文件操作时就会将这个指针以void *的形式传给文件服务进程，文件服务进程一开始就将其转化为了FileFd类型，并填充f_fileid与f_file(由于那一整页全部用来存Fd，所以可以肆无忌惮地利用后边的空间)
```

```
// 找 fd 对应的文件信息页面和文件缓存区地址
#define INDEX2FD(i) (FDTABLE + (i)*PTMAP)
#define INDEX2DATA(i) (FILEBASE + (i)*PDMAP)
```

关键信息

磁盘镜像

tools/fsformat.c

- 磁盘镜像制作工具
- `Block`与`disk`定义
- `Block`内字段`type`的枚举类型定义

文件操作库函数

user/include/fs.h

- 一些宏定义：

```
// 磁盘块大小
#define BLOCK_SIZE PAGE_SIZE
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8)

#define MAXNAMELEN 128 // filename最长值
#define MAXPATHLEN 1024 // pathname最长值

#define NDIRECT 10 // 直接指针个数
#define NINDIRECT (BLOCK_SIZE / 4) // 间接指针个数（前十个不可用）

#define MAXFILESIZE (NINDIRECT * BLOCK_SIZE) // 文件最大值

#define FILE_STRUCT_SIZE 256 // 文件控制块大小

#define FILE2BLK (BLOCK_SIZE / sizeof(struct File)) // 每个磁盘块最多含有的文件控制块个数，值为16

// 文件类型
#define FTYPE_REG 0 // Regular file
#define FTYPE_DIR 1 // Directory

#define FS_MAGIC // 文件魔数
```

- `Super`超级块的定义

```
struct Super {
    uint32_t s_magic;   // Magic number: FS_MAGIC; 用于验证文件系统的幻数
    uint32_t s_nblocks; // Total number of blocks on disk; 磁盘块总数
    struct File s_root; // Root directory node; 根目录文件节点
};
```

- `File` 结构体文件控制块的定义

```
struct File {
    char f_name[MAXNAMELEN]; // filename 文件名，最大长度为128
    uint32_t f_size;     // file size in bytes 文件大小，单位为字节
    uint32_t f_type;     // file type 文件类型，有普通文件和目录两种
    // 文件的直接指针，每个文件控制块设有10个直接指针，用来记录文件的数据块在磁盘上的位置
    uint32_t f_direct[NDIRECT];
    uint32_t f_indirect; // 文件的间接指针

    // 自己所在的目录
    struct File *f_dir; // the pointer to the dir where this file is in,
valid only in memory.
    // f_pad是为了让整数个文件结构体占用一个磁盘块，填充结构体中剩下的字节
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 -
sizeof(void *)];
} __attribute__((aligned(4), packed));
```

user/include/fd.h

- 一些宏定义

```
#define MAXFD 32 // 文件描述符数量最大值
#define FILEBASE 0x60000000 //
#define FDTABLE (FILEBASE - PDMAP) // 所有文件描述符位于[FILEBASE - PDMAP, FILEBASE)的地址空间中

#define INDEX2FD(i) (FDTABLE + (i)*PTMAP) // 通过文件描述符编号获取虚拟地址
#define INDEX2DATA(i) (FILEBASE + (i)*PDMAP)

// else  定义在include/mmu.h
#define PTMAP PAGE_SIZE
#define PDMAP (4 * 1024 * 1024) // bytes mapped by a page directory entry
```

- `file descriptor`文件描述符的定义

```
struct Fd {
    u_int fd_dev_id; // 文件对应的设备id
    u_int fd_offset; // 文件读写的偏移量
    u_int fd_omode;  // 文件读写的模式
};
```

- `Filefd`的定义

```
// file descriptor + file
struct Filefd {
    struct Fd f_fd; // 文件描述符
    u_int f_fileid; // 文件id
    struct File f_file; // 文件
};
```

- 结构体`struct Dev`的定义，实现了类似抽象类的功能

user/include/fsreq.h

对一系列从用户侧到文件服务管理进程的请求的描述

```
// Definitions for requests from clients to file system
FSREQ_OPEN, FSREQ_MAP, FSREQ_SET_SIZE, FSREQ_CLOSE,
FSREQ_DIRTY, FSREQ_REMOVE, FSREQ_SYNC
```

user/lib/file.c

定义了用户程序读写、创建、删除和修改文件的接口

- `fd_alloc`遍历所有文件描述符编号（共有`MAXFD`个），找到其中还没有被使用过的最小的一个，返回该文件描述符对应的地址。
- 定义了struct Dev类型的fevfile变量：

```
struct Dev devfile = {
    .dev_id = 'f',
    .dev_name = "file",
    .dev_read = file_read,
    .dev_write = file_write,
    .dev_close = file_close,
    .dev_stat = file_stat,
};
```

- 定义了一系列接口函数，如open、remove，本质上是去调用user/lib/fsipc.c中的fsipc_*函数用于与服务进程交互

user/lib/fsipc.c

- `fsipcipc`大小为`PAGESIZE`字节，以`PAGESIZE`对齐，所以是一个页面
- 该文件下有许多fsipc_*的函数，用于与服务进程交互，他们使用一个共同的接口——fsipc函数（也定义在user/lib/fsipc.c），该函数仅仅是向服务进程发生请求，然后接受服务进程传过来的数据
- 函数`int fsipc_map(u_int fileid, u_int offset, void *dstva)`: 将文件中offset所在的int read(iblock从磁盘映射到内存

user/lib/fd.c

- 函数`void *fd2data(struct Fd *fd)`: 获取文件内容应该映射到的地址

  整体的映射区间为`[FILEBASE, FILEBASE+1024*PDMAP)`。这正好在存储文件描述符的空间`[FILEBASE - PDMAP, FILEBASE)`的上面。
- 函数`int fd2num(struct Fd *fd)`: 获取文件描述符在文件描述符 “数组” 中的索引
- 函数`int read(int fdnum, void *buf, u_int n)`: 从fd中读取至多n个byte，存储到buf中，返回读取字节个数
- 函数`int fd_lookup(int fdnum, struct Fd **fd)`: 根据序号fdnum找到文件描述符
- `static struct Dev *devtab[]`: 全局设备表
- 函数`int dev_lookup(int dev_id, struct Dev **dev)`: 根据设备序号 dev_id 找到对应的设备
- 函数`int read(int fdnum, void *buf, u_int n)`: 从fd中读取至多n个byte，存储到buf中，返回读取字节个数

文件系统服务进程

> 文件系统服务进程是一个完整的进程，有自己的 main 函数。该进程的代码都位于 fs 文件夹下。`main`函数位于 fs/serv.c 中。

user/include/lib.h

定义了一些open modes

```
// File open modes
#define O_RDONLY 0x0000  /* open for reading only */
#define O_WRONLY 0x0001  /* open for writing only */
#define O_RDWR 0x0002    /* open for reading and writing */
#define O_ACCMODE 0x0003 /* mask for above modes */
#define O_CREAT 0x0100   /* create if nonexistent */
#define O_TRUNC 0x0200   /* truncate to zero length */
// O_TRUNC:如果该文件已经存在，文件的长度会被截断（即清空）为0字节。这通常用于需要重写整个文件内容的情况

// Unimplemented open modes
#define O_EXCL 0x0400  /* error if already exists */
#define O_MKDIR 0x0800 /* create directory, not regular file */
```

fs/serv.h

定义了和文件系统服务进程相关的宏

```
#define PTE_DIRTY 0x0004 // file system block cache is dirty
#define SECT_SIZE 512              /* Bytes per disk sector 扇区大小*/
#define SECT2BLK (BLOCK_SIZE / SECT_SIZE) /* sectors to a block 每个磁盘块扇区个数*/
// 将[DISKMAP,DISKMAP+DISKMAX)地址空间用作缓冲区，当磁盘读入内存时，用来映射相关的页。
#define DISKMAP 0x10000000
#define DISKMAX 0x40000000
```

可以通过DISKMAP+(n*BLOCK_SIZE)获得第n个磁盘块映射到的服务进程中的虚拟地址

fs/serv.c

- `opentab`与`Open`的定义

```
struct Open {
    struct File *o_file;
    u_int o_fileid;
    int o_mode;
    struct Filefd *o_ff;
};
#define MAXOPEN 1024
struct Open opentab[MAXOPEN];
```

- 一些宏定义

```
#define FILEVA 0x60000000 // 实际就是FILEBASE
#define REQVA 0x0ffff000 // 接收用户进程传来的页表的虚拟地址
```

- `serve_table`服务函数表
- 函数`main()`:文件系统服务进程的主函数，依次调用`serve_init()`，`fs_init()`，`serve()`。
- 函数`serve_init()`:对`opentab`进行初始化

之后我们介绍一下真正的用户请求处理函数:

> 前缀为`serve_`的函数是那些处理来自客户端的文件系统请求的函数。文件系统按功能接收请求`ipc_recv`，当接收到请求时，文件系统将调用相应的`serve_`，并通过函数`ipc_send`将结果返回给调用者。

- 函数`void serve(void)`:不断循环，通过ipc_recv接收用户进程的需求，然后转发给处理函数，即serve_table中的函数，实际是处理函数与用户间的接口
- 函数`void serve_open(u_int envid, struct Fsreq_open *rq)`: 依据rq中的路径打开文件，并存有FileFd的page通过ipc_send返回给用户
- 函数`int open_alloc(struct Open **o)`:申请一个open file

fs/fs.c

- 函数`fs_init()`, 完成对文件系统的初始化，依次调用`read_super`、`check_write_block`和`read_bitmap`
- 函数`read_super(void)`，读取读取对应磁盘块编号(其实就是1)的磁盘块数据到内存中读取对应磁盘块编号(其实就是1)的磁盘块数据到内存中，赋值给全局变量super
- 函数`int read_block(u_int blockno, void **blk, u_int *isnew)`,读取对应磁盘块编号的磁盘块数据到内存中
- 函数`int block_is_free(u_int blockno)`:Check if the block 'blockno' is free via bitmap.
- 函数`void *disk_addr(u_int blockno)`:返回磁盘块在内存中对应的虚拟地址
- 函数`void *block_is_mapped(u_int blockno)`:判断编号blockno对应的磁盘块是否已经被映射到一个真实的物理地址，返回其虚拟地址，没有则返回null，调用了函数`int va_is_mapped(void *va)`
- 函数`int va_is_mapped(void *va)`: 检查虚拟地址是否映射到一个磁盘块
- 函数`void read_bitmap(void)`:将管理磁盘块分配的位图读取到内存中
- 函数`int file_create(char *path, struct File **file)`: 创建这个路径，并将*file设置成这个路径的文件的指针
- 函数`int file_open(char *path, struct File **file)`:打开这个路径的文件，将*file设置成路径下文件的指针
- 函数`int walk_path(char *path, struct File **pdir, struct File **pfile, char *lastelem)`:

  解析路径，根据路径找到目录下的文件,并设置pdir与pfile与lastelem
- 函数`char *skip_slash(char *p)`:跳过斜杠'/'
- 函数`int dir_lookup(struct File *dir, char *name, struct File **file)`: 找到指定目录dir下的指定名字name的文件,写入*file
- 函数`int file_get_block(struct File *f, u_int filebno, void **blk)`:将某个指定的文件指向的磁盘块读入内存,获取文件中第filebno个磁盘块，写入*blk
- 函数`int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc)`: 获取文件f中使用的第filebno个磁盘块对应的磁盘块编号*diskbno,如果block未申请但是alloc非0的话则需要申请
- 函数`int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int alloc)`:

获取f中第filebno个磁盘块对应的disk中的磁盘块指针，如果indirect block之前未使用但是需要使用并且alloc非0时，将会申请indirect block

- 函数`int alloc_block(void)`:申请一个新的磁盘块,返回磁盘编号
- 函数`int alloc_block_num(void)`: 在磁盘块管理位图上找到空闲的磁盘块，更新位图并将位图写入内存,返回磁盘块编号
- 函数`void write_block(u_int blockno)`:将第blockno个磁盘块写回disk
- 函数`int map_block(u_int blockno)`： 检查指定的磁盘块是否已经映射到内存，如果没有，分配一页内存来保存磁盘上的数据
- 函数`void unmap_block(u_int blockno)`: 解除磁盘块和物理内存之间的映射关系，回收内存
- 函数`int block_is_dirty(u_int blockno)`:检查block是否被修改
- 函数`int va_is_dirty(void *va)`: 检查va所在页是否被修改
- 函数`int file_set_size(struct File *f, u_int newsize)`:设置文件大小
- 函数`void file_truncate(struct File *f, u_int newsize)`:将指定文件的大小截断为指定长度
- 函数`int file_clear_block(struct File *f, u_int filebno)`: 移除f中的第filebno个block

user/lib/pageref.c

只有`int pageref(void *v)`一个函数，用于得知某一页的引用数量；

这里`opentab`和 存储在`[FILEVA, FILEVA + PDMAP)`中的`Filefd`是一一对应的关系，所以通过查看`Filefd`地址的页表项是否有效。

但是为什么这里不能查看`opentab`中各元素的页表项呢，因为`opentab`作为数组，占用的空间已经被分配了。

磁盘驱动

> 实现设备读写系统调用 与 IDE磁盘读写
>
> 通过读写 PIIX4 的特定寄存器，我们可以实现以扇区为最小单元的读写。

kern/syscall_all.c

- 函数`int sys_write_dev(u_int va, u_int pa, u_int len)`: 将用户空间里[va, va+len) 的数据写到物理地址pa中
- 函数`int sys_read_dev(u_int va, u_int pa, u_int len)`: 将pa处的数据写入用户空间的[va, va+len) 中
- 函数`static inline int is_illegal_va_range(u_long va, u_int len)`: 判断地址段是否非法
- 函数`static inline int is_illegal_va(u_long va)`: 判断地址是否非法

include/io.h

实现了一系列io读写的函数，在内核态下使用

```
static inline uint8_t ioread8(u_long paddr); // 8 可以替换成 16 32
// uint8_t data = ioread8(pa); // read a byte at physical address `pa`.

static inline void iowrite8(uint8_t data, u_long paddr); // 8 可以替换成 16 32
//  iowrite8(data, pa); // write `data` to physical address `pa`.
```

include/malta.h

定义了与磁盘读写等相关的一系列宏

```
MALTA_IDE_DATA
MALTA_IDE_ERR
MALTA_IDE_NSECT
MALTA_IDE_LBAL
MALTA_IDE_LBAM
MALTA_IDE_LBAH
MALTA_IDE_DEVICE
MALTA_IDE_STATUS
MALTA_IDE_LBA
MALTA_IDE_BUSY
MALTA_IDE_CMD_PIO_RE /* Read sectors with retry */
MALTA_IDE_CMD_PIO_WRIT /* write sectors with retry */
.................
```

表 5.2: PIIX4 I/O 关键寄存器映射

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

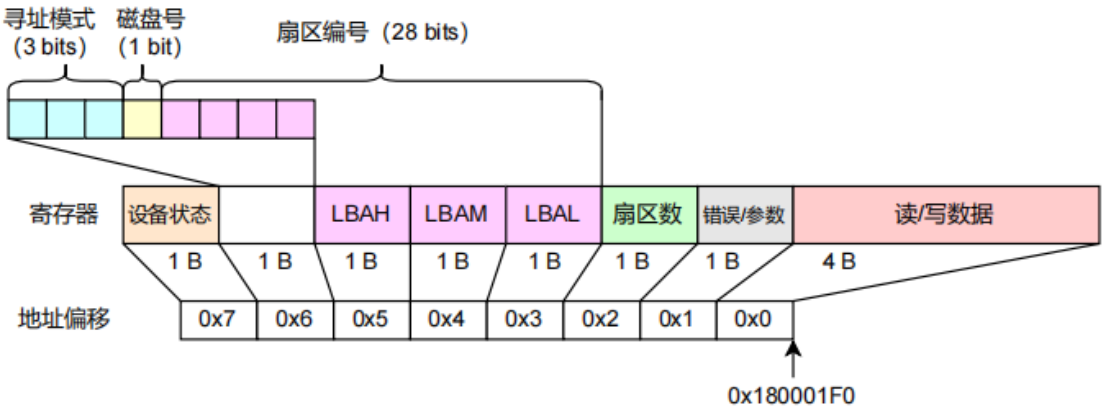


图 5.3: LBA 模式下参数和寄存器映射关系图

fs/ide.c

通过调用之前的系统调用，实现了磁盘的读写操作

- 函数`static uint8_t wait_ide_ready()`：等待IDE设备准备好服务当前请求
- 函数`void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)`：从IDE磁盘读取数据

  用法：ide_read(0, blockno * SECT2BLK, va, SECT2BLK)
- 函数`void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)`：将数据写入IDE磁盘

  用法：ide_write(0, blockno * SECT2BLK, va, SECT2BLK)