BUAA_OS_Lab5笔记

重要宏

`FILE2BLK` 表示在一个磁盘块（block）中最多可以容纳多少个 `struct File` 文件项。即：一个文件最多可以用到 `FILE2BLK` 个块（block），即最多可以容纳这么多个子文件或目录项。**遍历目录模板**时会使用。

IDE磁盘驱动

位于用户空间，对于设备的读写必须通过系统调用来实现。

我们采用逻辑块寻址的方式来进行扇区寻址。IDE 设备将磁盘看作一个线性的字节序列，每个扇区都有一个唯一的编号，只需要设置目标扇区编号，就可以完成磁盘的寻址。在实验中，扇区编号有 28 位，每个扇区 512 B，因此最多可以有 128 GB 的磁盘空间。

基本概念：

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

`kern/syscall_all.c/sys_write_read_dev`

实现设备的读写操作。在实验中允许访问的物理地址范围仅有2段（console 和 IDE disk）。

`iowrite` 和 `ioread` 系列函数定义于 `io.h`，用于从物理地址中读写数据。

`fs/ide.c/ide_read ide_write`

用于在用户态下**读写磁盘**。

当前目录下检查 IDE 状态的帮手函数 `wait_ide_ready`，用于等待 IDE 上一个动作完成。

由于一次要读 **512 字节**（即一个扇区），IDE 数据端口（`MALTA_IDE_DATA`）**每次只能传输4个字节**（32位），这是硬件规定。所以要把512字节的数据分成 **128次，每次读4字节。**

```
// Step 8: 读取数据本体
for (int i = 0; i < SECT_SIZE / 4; i++) {
    panic_on(syscall_read_dev(dst + offset + i * 4, MALTA_IDE_DATA, 4));
}

// Step 9: 读取状态寄存器以确认是否读成功
panic_on(syscall_read_dev(&temp, MALTA_IDE_STATUS, 1));

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

文件系统结构设计

使用位图法来管理空闲的磁盘资源，用一个二进制位 bit 标识磁盘中的一个磁盘块的使用情况（**1 表示空闲，0 表示占用**）。

`user/include/fs.h`

`user/include/fs.h` 中定义了 `File` 结构体（**文件控制块**）、`Super` 结构体、相关宏等。

```
struct File {
    char f_name[MAXNAMELEN]; // filename
    uint32_t f_size;     // file size in bytes
    uint32_t f_type;     // file type
    uint32_t f_direct[NDIRECT];
    uint32_t f_indirect;

    struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory.
    char f_pad[FILE_STRUCT_SIZE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

`tools/fsformat.c`

`init_disk`

磁盘块数据结构：

```
struct Block {
    uint8_t data[BLOCK_SIZE]; // 存储内容：可能是位图，也可能是文件数据、索引等
    uint32_t type;            // 表示这个块是什么类型（enum中，常见有根、超级块、位图块等）
} disk[NBLOCK];
```

```
#define NBLOCK 1024          // 磁盘的总块数
#define BLOCK_SIZE 4096      // 每块 4096 字节（来自 user/include/fs.h）
#define BLOCK_SIZE_BIT (BLOCK_SIZE * 8) // 每块的比特数
```

Block No	用途
0	引导块（boot sector），由系统保留，不可重写或释放
1	超级块（super block）
2+	位图块
`nextbno` 起	用户数据、文件块、索引块等

`*create_file`

在指定目录 `dirf` 中找一个空闲 `struct File` 项目来创建新文件，**返回一个未使用的 File 结构体指针。**

经典的 “**直接块 + 间接块**” 文件块索引方式：

```
if (i < NDIRECT) {
    bno = dirf->f_direct[i];
} else {
    bno = ((int*)(disk[dirf->f_indirect].data))[i];
}
```

`fs/fs.c`

文件系统管理模块（File system），用于磁盘块管理、文件块映射、目录结构、路径解析、文件读写同步等所有逻辑。

`disk_addr`

根据一个块的序号 (blockno)，计算这一磁盘块对应的虚存的起始地址。

`map_block`

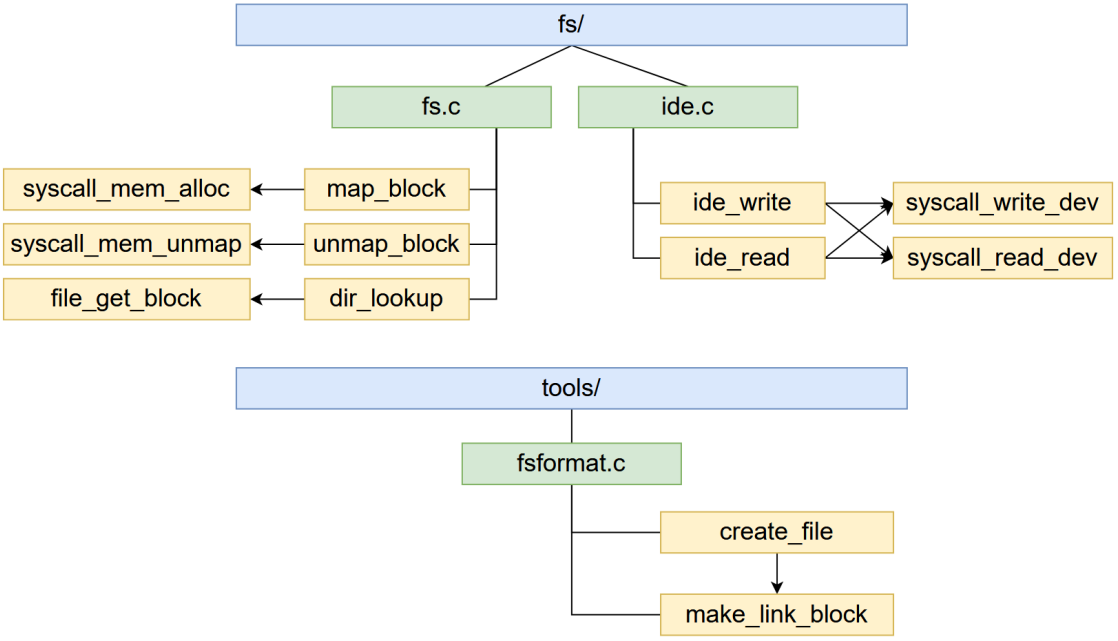
把一个磁盘块中的内容载入到内存中时，需要为之分配对应的物理内存。

`unmap_block`

结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。

`dir_lookup`

查找某个目录下是否存在指定的文件。



文件系统的用户接口

文件系统属于用户态进程。文件描述符（**f**ile **d**escriptor）是用户程序管理、操作文件的基础。换句话说，**凡是涉及到 `fd` 的全都处于用户态。**

`user/include/fd.h` 中定义了两个非常重要的结构体：

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

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

典型使用方式例如 `fd` 无法直接获得文件大小，但通过：

```
// user/lib/file.c/open
ffd = (struct Filefd*)fd;
size = ffd->f_file.f_size;
```

就可以通过强制类型转换成 `Filefd` ，通过其中 `File` 类型的元素 `f_file` 来获取文件大小。

`user/lib/file.c`

`open`

若成功打开文件，则该函数返回文件描述符的编号。

`remove`

删除文件或目录。

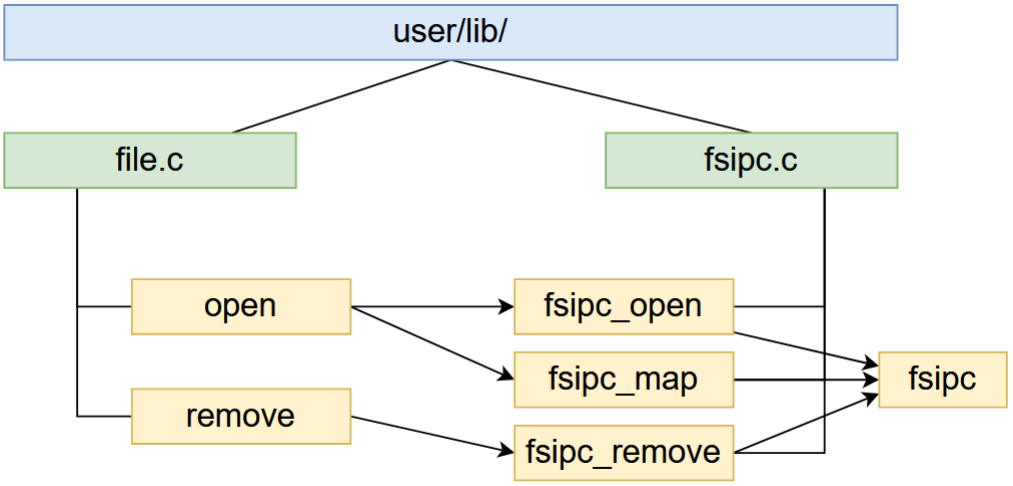
`user/lib/fd.c/read`

实现文件读取。

`fs/serv.c/serve_remove`

`user/lib/fsipc.c/fsipc_remove`

上面2个函数和 `user/lib/file.c/remove` 联合起来共同实现删除指定路径的文件。



查询目录内容

功能简述：在用户程序中调用 `fs_listdir(path)`，由 `fsipc` 发出 IPC 请求，server 端接收请求，查询对应目录的文件列表，并返回用户态。

文件	作用	需要做的修改
`user/include/fsreq.h`	定义系统调用号和请求结构体	添加 `FSREQ_LISTDIR` 常量和请求结构
`user/lib/fsipc.c`	用户态封装 IPC 请求	添加 `fsipc_listdir` 函数
`user/lib/file.c`	用户态接口	添加 `fs_listdir(const char *path)` 函数

文件	作用	需要做的修改
`fs/serv.c`	文件服务器主程序	添加对 `FSREQ_LISTDIR` 的处理

1. 🔧 `fsreq.h`：**定义请求结构与请求号（协议）**

在已有的 `enum` 后（注意是 `MAX_FSREQNO` 之前，注意**逗号**）添加一个新的请求号：

```
FSREQ_LISTDIR,   // 新增
```

添加请求结构体（要知道为什么这里需要添加2个请求结构体，而课下实验中均只需要1个的原因）：

```
struct Fsreq_listdir {
    char req_path[MAXPATHLEN];  // 目录路径
};

struct Fsret_listdir {
    int count;                  // 返回的文件数量
    char names[FILE2BLK][MAXNAMELEN]; // 文件名列表（最多一个块的量）
};
```

2. 🧠 `fsipc.c`：**添加客户端封装函数**

可以发现，**用户态的 `fsipc.c` 和内核态的 `serv.c` 中的函数几乎是一一对应的。** `fsipc.c` 最终都会调用 `fsipc` 函数，用于给 server 发出文件请求。

注意因为文件系统服务端是**通过页共享（ipc_send）将数据页返回给客户端的**，必须：

- 告诉内核我们需要在 `retbuf` 这个地址接收一个页
- 同时**接收这个页的权限信息**，以确保共享成功、权限正确

```
int fsipc_listdir(const char *path, struct Fsret_listdir *retbuf) {
    struct Fsreq_listdir *req = (struct Fsreq_listdir *)fsipcbuf;

    if (strlen(path) == 0 || strlen(path) >= MAXPATHLEN) {
        return -E_BAD_PATH;
    }

    strcpy(req->req_path, path);
    u_int perm;
    return fsipc(FSREQ_LISTDIR, req, retbuf, &perm);
}
```

3. 📦 `file.c`：**添加用户态 `listdir()` 封装**

```c
int listdir(const char *path, char names[FILE2BLK][MAXNAMELEN], int *count)
{
    struct Fsret_listdir *retbuf = (struct Fsret_listdir *)fsipcbuf;

    int r = fsipc_listdir(path, retbuf);
    if (r < 0) return r;

    *count = retbuf->count;
    for (int i = 0; i < retbuf->count; i++) {
        strcpy(names[i], retbuf->names[i]);
    }

    return 0;
}
```

或者（如果不暴露 `fsipcbuf`）：

```c
int listdir(const char *path, char names[FILE2BLK][MAXNAMELEN], int *count)
{
    struct Fsret_listdir retbuf;

    int r = fsipc_listdir(path, &retbuf);  // 注意：retbuf 是局部变量
    if (r < 0) return r;

    *count = retbuf.count;
    for (int i = 0; i < retbuf.count; i++) {
        strcpy(names[i], retbuf.names[i]);
    }

    return 0;
}
```

注意不论哪种写法，此处都没有引入头文件，因此不能直接使用 `fsipc_listdir`。具体情况要看题目要求。

4. 🖥️ `serv.c`：添加服务端目录查询服务

✅ 在 serve_table[] 中添加注册：

```c
[FSREQ_LISTDIR] = serve_listdir,
```

✅ 添加 `serve_listdir()` 函数：

```c
void serve_listdir(u_int envid, struct Fsreq_listdir *rq) {
    int r;
    struct File *dir;
    void *blk;
    struct Fsret_listdir *ret = (struct Fsret_listdir *)REQVA;

    // Step 1: Try to open the path as a directory.
    if ((r = file_open(rq->req_path, &dir)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
```

```
    // Step 2: Confirm it's actually a directory.
    if (dir->f_type != FTYPE_DIR) {
        ipc_send(envid, -E_NOT_FOUND, 0, 0);
        return;
    }

    // Step 3: Traverse directory blocks and collect names.
    int nblock = dir->f_size / BLOCK_SIZE;
    int count = 0;

    for (int i = 0; i < nblock && count < FILE2BLK; i++) {
        if ((r = file_get_block(dir, i, &blk)) < 0) {
            ipc_send(envid, r, 0, 0);
            return;
        }

        struct File *files = (struct File *)blk;
        for (int j = 0; j < FILE2BLK && count < FILE2BLK; j++) {
            if (files[j].f_name[0] != '\0') {
                strcpy(ret->names[count++], files[j].f_name);
            }
        }
    }

    // Step 4: Set result count and send back the page.
    ret->count = count;
    ipc_send(envid, 0, ret, PTE_D | PTE_LIBRARY);
}
```

其中**二重遍历目录**开源仓库中一般这么写（`tools/fsformat.c/*create_file`、`fs/fs.c/dir_lookup`），可以看实际需求：

```
    // Iterate through all existing blocks in the directory.
    int nblk = dirf->f_size / BLOCK_SIZE;       // 模板，牢记
    for (int i = 0; i < nblk; ++i) {
        // ...

        // Iterate through all 'File's in the directory block.
        for (struct File *f = blk; f < blk + FILE2BLK; ++f) {
            if (f->f_name[0] != '\0') {
                // ...
            }
        }
    }
```

支持可选加密

🧭 功能目标概述

- 支持对某些文件进行加密存储，控制方式由 `open` 的一个新标志 `O_ENCRYPT` 控制。
- 加密文件的内容在写入时被加密，在读取时解密。
- 支持通过 `setkey()` 设置全局密钥，并存储到文件系统中一个固定块。

- 非加密文件照常处理；加密文件自动加解密。

文件	内容
`fs.h`	添加 `O_ENCRYPT` 和 `setkey` 接口声明
`fs.c`	添加 `set_encryption_key`, `load_encryption_key`, `encrypt_decrypt_buffer`
`file.c`	修改 `file_read`, `file_write` 实现加解密逻辑，添加 `setkey`
`fsreq.h`	添加 `FSREQ_SETKEY` 和 `Fsreq_setkey` 结构体
`fsipc.c`	添加 `fsipc_setkey()`
`serv.c`	添加 `serve_setkey()` 并注册到 `serve_table`

第一步：统一添加 `O_ENCRYPT` 标志

修改：`user/include/lib.h`，加在已有的 `O_RDONLY` 等 `File open modes` 宏定义之后：

```
#define O_ENCRYPT 0x800  // 自定义文件加密标志位
```

第二步：密钥存储

修改`fs.c`，新增全局变量和密钥加载函数：

```
#define ENCRYPT_KEY_BLOCKNO 2   // 假设使用块 2 存储密钥，这里操作具体看要求
#define ENCRYPT_KEY_LEN 32

static char encryption_key[ENCRYPT_KEY_LEN];    // 看情况要不要静态
static int key_valid = 0;

int set_encryption_key(const char *key) {
    if (strlen(key) >= ENCRYPT_KEY_LEN) return -E_INVAL;
    strcpy(encryption_key, key);
    key_valid = 1;

    void *blk;
    if (read_block(ENCRYPT_KEY_BLOCKNO, &blk, 0) < 0) return -E_NO_DISK;
    memset(blk, 0, BLOCK_SIZE);
    strcpy((char *)blk, encryption_key);
    write_block(ENCRYPT_KEY_BLOCKNO);
    return 0;
}

int load_encryption_key() {
    void *blk;
    if (read_block(ENCRYPT_KEY_BLOCKNO, &blk, 0) < 0) return -E_NO_DISK;
    strncpy(encryption_key, (char *)blk, ENCRYPT_KEY_LEN - 1);
    key_valid = 1;
    return 0;
}
```

第三步：添加加解密函数（异或示意）

```c
void encrypt_decrypt_buffer(void *buf, u_int len) {
    if (!key_valid) return;
    for (u_int i = 0; i < len; ++i) {
        ((char *)buf)[i] ^= encryption_key[i % strlen(encryption_key)];
    }
}
```

第四步：加密写入、解密读取 hook

在`file.c`中：

修改`file_read()`：

```c
// 原 memcpy 后添加：
memcpy(buf, (char *)fd2data(fd) + offset, n);

if (fd->fd_omode & O_ENCRYPT) {
    encrypt_decrypt_buffer(buf, n);  // 解密
}
```

修改`file_write()`：

```c
// 修改最后 Write the data 部分
if (fd->fd_omode & O_ENCRYPT) {
    char temp[MAXFILESIZE];
    memcpy(temp, buf, n);
    encrypt_decrypt_buffer(temp, n);  // 加密
    memcpy((char *)fd2data(fd) + offset, temp, n);
} else {
    memcpy((char *)fd2data(fd) + offset, buf, n);
}
```

第五步：支持用户调用设置密钥

从这里往下是非常套路化的 IPC 机制用户级线程通信请求：

目标是新增`int sys_setkey(const char *key);`

用户接口头文件`user/include/lib.h`：

```c
int setkey(const char *key);  // 用户态接口
```

用户态接口`file.c`中新增：

```c
int setkey(const char *key) {
    return fsipc_setkey(key);  // 走 IPC
}
```

第六步：IPC 支持设置密钥请求

修改：`fsreq.h`

```c
#define FSREQ_SETKEY 8  // 新增到 enum 后面

struct Fsreq_setkey {
    char key[ENCRYPT_KEY_LEN];
};
```

第七步：实现 `fsipc_setkey()`

修改：`fsipc.c`

```c
int fsipc_setkey(const char *key) {
    struct Fsreq_setkey *req = (struct Fsreq_setkey *)fsipcbuf;
    strncpy(req->key, key, ENCRYPT_KEY_LEN);
    return fsipc(FSREQ_SETKEY, req, 0, 0);
}
```

第八步：server 添加 `serve_setkey()`

修改：`serv.c`

```c
void serve_setkey(u_int envid, struct Fsreq_setkey *rq) {
    int r = set_encryption_key(rq->key);
    ipc_send(envid, r, 0, 0);
}
```

第九步：连接 `serve_table`

```c
[FSREQ_SETKEY] = serve_setkey,
```

这里补充一下：如果还需要在 `file/open()` 函数中 `fsipc_open` 成功后**加载密钥**：

```c
// Step 2: Prepare the 'fd' using 'fsipc_open' in fsipc.c.
r = fsipc_open(path, mode, fd);
if (r) return r;

// 尝试加载密钥（如果之前 setkey 成功，key_valid 会是 1）
fsipc_loadkey(); // 确保在 fsipc.c 实现了这个函数

// 后续代码不变
```

也可以直接调用 `load_encryption_key()`，只要能让它从 `fsipc` 层走 IPC 请求，和刚才的 `setkey()` 做法类似。都是**走 IPC 通道的三部曲**：`serv.c`、`fsipc.c`、`file.c`，和指导书最后一套任务完全相同。给出三部曲的具体实现：

用户态调用接口，也就是从"第五步"开始：

用户接口头文件 `user/include/lib.h`：

```c
int loadkey();  // 用户态接口
```

用户态接口 `file.c` 中新增：

```
int loadkey() {
    return fsipc_loadkey(key);  // 走 IPC
}
```

IPC 通信机制，主要涉及到 `fsreq`、`fsipc` 等等：

在 `fsreq.h` 的 `enum` 中添加新的请求类型：

```
    FSREQ_LOADKEY, // ← 新增

// 然后实现：
struct Fsret_loadkey {
    char key[ENCRYPT_KEY_LEN];
};
```

在 `fsipc.c` 中添加如下函数：

```
int fsipc_loadkey() {
    struct Fsret_loadkey *ret = (struct Fsret_loadkey *)fsipcbuf;
    u_int perm;

    int r = fsipc(FSREQ_LOADKEY, NULL, ret, &perm);
    if (r < 0) {
        return r;
    }

    // 将密钥写入 file.c 中的缓存
    extern char encryption_key[];
    extern int key_valid;
    if (ret->key_len > 0 && ret->key_len <= ENCRYPT_KEY_LEN) {
        memcpy(encryption_key, ret->key, ret->key_len); //这句是核心
        encryption_key[ret->key_len] = '\0';  // 确保末尾安全
        key_valid = 1;                                  //这句是核心
    } else {
        key_valid = 0;
        return -E_INVAL;
    }

    return 0;
}

// 或者：
int fsipc_loadkey(struct Fsret_loadkey *retbuf) {
    return fsipc(FSREQ_LOADKEY, NULL, retbuf, NULL);
}
```

注意：如果把 `encryption_key` 和 `key_valid` 放在 外面（之前是 `fd`），你需要在 `fsipc.c` 用 `extern` 声明（如果是静态该方法行不通）。

服务端，就是 server：

在 `serv.c`（文件服务器）中：

```
void serve_loadkey(u_int envid, u_int reqva) {
```

```
    struct Fsret_loadkey *ret = (struct Fsret_loadkey *)reqva;

    // 用一个本地缓冲区读取一个完整的磁盘块
    char temp_block[BLOCK_SIZE];

    // 用 ide_read 从磁盘中读取 ENCRYPT_KEY_BLOCKNO 所在块
    int r = ide_read(ENCRYPT_KEY_BLOCKNO * SECT_SIZE, temp_block,
SECT_SIZE);
    if (r < 0) {
        ipc_send(envid, r, 0, 0); // 如果读取失败，直接返回错误
        return;
    }

    // 将前 ENCRYPT_KEY_LEN 字节拷贝进返回结构体
    memcpy(ret->key, temp_block, ENCRYPT_KEY_LEN);
    ret->key_len = ENCRYPT_KEY_LEN;

    // 通过 ipc_send 发送结构体回客户端
    ipc_send(envid, 0, ret, PTE_D | PTE_LIBRARY);
}
```

在 `serve` 中注册处理器：

```
[FSREQ_LOADKEY] = serve_loadkey,
```