一、Lab5 前言

这是最长的一篇文章，可就算这么长，文中出现的代码也不过本次 Lab 中新增加的代码的一小部分。幸好完成本次实验不需要熟悉所有代码，一部分练习甚至不需要熟悉要填写的代码的前后文，只需要根据注释就可以填出很多。可是我感觉本篇文章还是有帮助的，毕竟谁也不知道 Exam 会出什么题。

Lab5 主要分为四部分，分别是镜像制作工具、关于设备的系统调用、文件系统服务进程、文件操作库函数。本文对这四个方面都有所涉及，第二章主要讲镜像制作工具，第三章主要讲文件系统服务进程和文件操作库函数，最后一章讲关于设备的系统调用。

二、磁盘镜像

（1）镜像制作工具

在本次实验中我们要实现一个文件系统。广义来说，一切字节序列都可以称为文件，但本次实验中我们还是主要关注在磁盘中存储的数据，将这些数据按一定的结构组织起来，就是本次实验的主要目标。

本文依旧不按照指导书中的顺序。我们先查看位于 tools 文件夹下的磁盘镜像制作工具 fsformat 的源代码，以便我们理解磁盘以及文件系统的组织结构。

（2）磁盘数据初始化

我们查看 tools/fsformat.c 文件。找到其中的 `main` 函数。`main` 函数首先调用了 `init_disk` 用于初始化磁盘。

```
int main(int argc, char **argv) {
    static_assert(sizeof(struct File) == BY2FILE);
    init_disk();
```

该函数中我们要用到一个数据结构 `disk`。因此我们先考察 `disk`。`disk` 是一个数组，大小为 `NBLOCK`，每个元素是一个结构体，其中有字段 `data`，是一个 `BY2BLK` 字节大小的空间，用于存储一个磁盘块的数据。很容易得知，`NBLOCK` * `BY2BLK` = 磁盘空间大小。这样就可以理解 `disk` 起到的作用了，也就是在构筑磁盘镜像时暂时存储磁盘数据，等到构筑完成后再将 `disk` 中 `data` 的内容拼接并输出为二进制镜像文件。

```
struct Block {
    uint8_t data[BY2BLK]; // 4096
    uint32_t type;
} disk[NBLOCK]; // 1024
```

> 磁盘块是对磁盘空间的逻辑划分；扇区是对磁盘空间的物理划分

另外 `Block` 结构体还有一个字段 `type`，该字段的值为如下枚举的值

```
enum {
    BLOCK_FREE = 0,
    BLOCK_BOOT = 1,
```

```
    BLOCK_BMAP = 2,
    BLOCK_SUPER = 3,
    BLOCK_DATA = 4,
    BLOCK_FILE = 5,
    BLOCK_INDEX = 6,
};
```

让我们回到 `init_disk`。该函数中首先将第一个磁盘块类型设为 `BLOCK_BOOT`，表示主引导扇区。之后我们要从第三个磁盘块开始（为什么不是第二个？因为第二个磁盘块为 “超级块”，将在后面介绍），设置磁盘块的位图分配机制。在函数中我们计算了在磁盘中存储位图需要的磁盘块数量。`NBLOCK` 是磁盘块的总数，那么我们同样需要 `NBLOCK` bit 大小的位图，又因为一个磁盘块有 `BIT2BLK` bit，那么总共需要 `NBLOCK / BIT2BLK` 个磁盘块。向上取整，总共需要 `(NBLOCK + BIT2BLK - 1) / BIT2BLK` 个磁盘块来存储位图。现在我们已经将 0 到 nbitblock-1 的位图分配了用途，那么下一个空闲的磁盘块就是 `nextbno = 2 + nbitblock` 了。

```
    // Step 2: Initialize boundary.
    nbitblock = (NBLOCK + BIT2BLK - 1) / BIT2BLK;
    nextbno = 2 + nbitblock;
```

对于存储位图的磁盘块，我们要将其初始化。首先我们将这些磁盘块标记为 `BLOCK_BMAP`，表示他们用作存储位图

```
    // Step 2: Initialize bitmap blocks.
    for (i = 0; i < nbitblock; ++i) {
        disk[2 + i].type = BLOCK_BMAP;
    }
```

对于位图，我们设定 1 表示空闲，0 表示使用。因此我们先将所有的磁盘块数据都设定为 1。

```
    for (i = 0; i < nbitblock; ++i) {
        memset(disk[2 + i].data, 0xff, BY2BLK);
    }
```

最后如果位图不足以占用全部空间，那么我们还需要将最后一个磁盘块末位不作为位图使用的部分置 0。

```
    if (NBLOCK != nbitblock * BIT2BLK) { // nbitblock 是向上取整了
        diff = NBLOCK % BIT2BLK / 8;
        memset(disk[2 + (nbitblock - 1)].data + diff, 0x00, BY2BLK - diff);
    }
```

我们不要忘记了第二个磁盘块，这个磁盘块会用作 “超级块”，所谓超级块，就是文件系统的起点，该磁盘块中存储了根目录文件的信息（当然还包括其他一些内容）。超级块定义在 user/include/fs.h 中。包含了用于验证文件系统的幻数 `s_magic`，磁盘的磁盘块总数 `s_nblocks` 和根目录文件节点 `s_root`。

```
struct Super {
    uint32_t s_magic;   // Magic number: FS_MAGIC
    uint32_t s_nblocks; // Total number of blocks on disk
    struct File s_root; // Root directory node
};
```

在本文件系统中，文件信息通过 `struct File` 结构体存储。该结构体同样定义在 user/include/fs.h 总

```
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[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
} __attribute__((aligned(4), packed));
```

该结构体中包含了文件名 `f_name`，文件大小 `f_size`，文件类型 `f_type`，和用于存储指向存储文件内容的磁盘块的编号的数组 `f_direct`，以及指向存储 “存储指向存储文件内容的磁盘块的编号的数组” 的磁盘块的编号 `f_indirect`，还有自己所在的目录 `f_dir`。最后 `f_pad` 将 `struct File` 结构体的大小填充到 `BY2FILE`，保证多个 `struct File` 能够填满整个磁盘。

> `f_direct` 和 `f_indirect` 用一句话来表示似乎有些困难，这里再重新说明一下。文件的内容需要在磁盘中存储，这些内容分布于不同的磁盘块中，因此还需要对这些内容进行管理，也就是要再分配一个磁盘块用于存储 `struct File`，该结构体中要存储哪些存储文件内容的磁盘块的地址。`f_direct` 中就存储了前 `NDIRECT` 个磁盘块的编号方便快速访问。但如果文件较大，超出了 `NDIRECT` 个磁盘块的大小的话要怎么办？这就要再分配一个磁盘块，用这个磁盘块的全部空间作为存储磁盘块编号的数组，保存那些存储了文件内容的磁盘块的编号。再使用 `f_indirect` 保存这个新分配的，作为数组的磁盘块的编号。
>
> 为了方便，`f_indirect` 指向的磁盘块的前 `NDIRECT` 个元素不保存编号。现在我们就可以计算在这个文件系统中，文件的最大大小了：我们可以有 `BY2BLK / 4` 个磁盘块用于存储，每个磁盘块又有 `BY2BLK` 的空间，那么总共就有 `BY2BLK**2 / 4` 的空间可以用于单一文件的存储。

在 `dist_init` 的最后，就设置了超级块的信息，

```
    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, "/");
}
```

其中设置了根目录文件类型 `s_root.f_type` 为 `FTYPE_DIR`，表示该文件为目录。此宏定义在 user/include/fs.h 中。类似还有 `FTYPE_REG`。

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

最后我们设置根目录的名字为 `/`，这说明该文件系统的路径是 Linux 的格式。

（3）文件写入

在 `main` 函数的后面部分，我们不断读取命令行参数，通过 `stat` 函数（这是库函数）判断文件类型，分别调用 `write_directory` 和 `write_file` 将文件内容写入磁盘镜像中。需要注意这里传入了 `&super.s_root`，也就是根目录文件作为参数。

```
    if (argc < 3) {
        fprintf(stderr, "Usage: fsformat <img-file> [files or directories]...\n");
        exit(1);
    }

    for (int i = 2; i < argc; i++) {
        char *name = argv[i];
        struct stat stat_buf;
        int r = stat(name, &stat_buf);
        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 {
            fprintf(stderr, "'%s' has illegal file mode %o\n", name,
stat_buf.st_mode);
            exit(2);
        }
    }
```

接下来我们查看 `write_directory` 和 `write_file`。`write_directory` 用于递归地将目录下所有文件写入磁盘，而 `write_file` 则只将单一文件写入。我们首先查看 `write_directory`。

一开始只是调用库函数 `opendir` 打开了目录，不需关注。第一个关注点是调用了 `create_file` 在 `dirf` 文件下创建了新文件。

```
void write_directory(struct File *dirf, char *path) {
    DIR *dir = opendir(path);
    if (dir == NULL) {
```

```
        perror("opendir");
        return;
    }
    struct File *pdir = create_file(dirf);
```

在 create_file 函数中，遍历了 dirf 文件下用于保存内容（对于目录来说，内容就是 struct File 结构体）的所有磁盘块。这里我们使用 f_direct 和 f_indirect 获取了对应磁盘块的编号。

```
struct File *create_file(struct File *dirf) {
    int nblk = dirf->f_size / BY2BLK;

    // Step 1: Iterate through all existing blocks in the directory.
    for (int i = 0; i < nblk; ++i) {
        int bno; // the block number
        // If the block number is in the range of direct pointers (NDIRECT), get
the 'bno'
        // directly from 'f_direct'. Otherwise, access the indirect block on
'disk' and get
        // the 'bno' at the index.
        /* Exercise 5.5: Your code here. (1/3) */
        if (i < NDIRECT) {
            bno = dirf->f_direct[i];
        } else {
            bno = ((uint32_t *)(disk[dirf->f_indirect].data))[i];
        }
```

该磁盘块的空间全部用于存储 struct File。我们再在一个磁盘块中遍历所有 struct File，看是否有未使用的 struct File，用该处空间表示我们新创建的文件。这样遍历的原因是可能出现中间磁盘块中文件被删除，或者最后一个磁盘块还未用满的情况。

```
        // Get the directory block using the block number.
        struct File *blk = (struct File *)(disk[bno].data);

        // Iterate through all 'File's in the directory block.
        for (struct File *f = blk; f < blk + FILE2BLK; ++f) {
            // If the first byte of the file name is null, the 'File' is unused.
            // Return a pointer to the unused 'File'.
            /* Exercise 5.5: Your code here. (2/3) */
            if (f->f_name[0] == NULL) {
                return f;
            }
        }
    }
```

最后如果没有找到未使用的 struct File，就说明所有的已分配给当前目录文件，用于存储 struct File 的磁盘块都被占满了。这时就需要调用 make_link_block 新申请一个磁盘块。该磁盘块中第一个 struct File

的空间就是新创建的文件。

```
    // Step 2: If no unused file is found, allocate a new block using
'make_link_block' function
    // and return a pointer to the new block on 'disk'.
    /* Exercise 5.5: Your code here. (3/3) */
    return (struct File *)(disk[make_link_block(dirf, nblk)].data);
```

`make_link_block` 很简单，就是获取下一个空闲的磁盘块，调用 `save_block_link` 为 `dirf` 目录文件添加该磁盘块。

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

int make_link_block(struct File *dirf, int nblk) {
    int bno = next_block(BLOCK_FILE);
    save_block_link(dirf, nblk, bno);
    dirf->f_size += BY2BLK;
    return bno;
}
```

`save_block_link` 函数就是将新申请的磁盘块设置到 `f_direct` 中或 `f_indirect` 对应的磁盘块中的相应位置。

```
void save_block_link(struct File *f, int nblk, int bno) {
    assert(nblk < NINDIRECT); // if not, file is too large !

    if (nblk < NDIRECT) {
        f->f_direct[nblk] = bno;
    } else {
        if (f->f_indirect == 0) {
            // create new indirect block.
            f->f_indirect = next_block(BLOCK_INDEX);
        }
        ((uint32_t *)(disk[f->f_indirect].data))[nblk] = bno;
    }
}
```

这样 `create_file` 就完成了，让我们回到 `write_directory`。接下来为新创建的目录文件设置名字和文件类型。途中还判断了文件名是否过长。

```
    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;
```

接下来的步骤很明显，需要遍历宿主机上该路径下的所有文件，如果是目录，则递归执行 `write_directory`，如果是普通文件，则执行 `write_file`，这样直到目录下所有文件都被写入镜像中。

```
    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 {
                write_file(pdir, buf);
            }
            free(buf);
        }
    }
    closedir(dir);
}
```

接下来转过头查看一下 `write_file`。首先我们同样调用 `create_file` 在 `dirf` 目录文件下创建新文件。

```
void write_file(struct File *dirf, const char *path) {
    int iblk = 0, r = 0, n = sizeof(disk[0].data);
    struct File *target = create_file(dirf);
```

接下来函数还对 `create_file` 的结果进行了判断。这似乎只是修改代码后的一个遗留问题而已。

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

我们打开宿主机上的文件，便于后面复制文件内容到镜像中

```
    int fd = open(path, O_RDONLY);
```

接着我们复制文件名。这里使用 `strrchr` 从后往前查找了 `'/'` 字符的位置，只拷贝该字符之后的内容。但是不知这里为何不与 `write_directory` 一样使用 `basename`。

```
    // Get file name with no path prefix.
    const char *fname = strrchr(path, '/');
    if (fname) {
        fname++;
    } else {
        fname = path;
    }
    strcpy(target->f_name, fname);
```

接着使用 `lseek` 获取并设置文件大小，以及文件类型为普通文件

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

最后读取文件内容，写入镜像文件中。这里我们以 `n = sizeof(disk[0].data)` 的大小读取，但不知道为何不使用 `BY2BLK`。值得注意的是，这里我们是先向 `disk[nextbno]` 中写入了数据，之后才调用 `next_block` 申请的该磁盘块。

```
    // Start reading file.
    lseek(fd, 0, SEEK_SET);
    while ((r = read(fd, disk[nextbno].data, n)) > 0) {
        save_block_link(target, iblk++, next_block(BLOCK_DATA));
    }
    close(fd); // Close file descriptor.
}
```

（4）收尾工作

完成文件的写入后，`main` 函数中还有一些收尾工作。首先是根据磁盘块的使用情况设置位图

```
    flush_bitmap();
```

置 0 语句看似复杂，实际只是按顺序一位一位置 0 而已。

```
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 / BIT2BLK].data)[(i % BIT2BLK) / 32] &= ~(1 << (i
% 32));
```

```
    }
}
```

最后还要根据 `disk` 生成磁盘镜像文件。使用 `finish_fs` 函数完成

```

    finish_fs(argv[1]);

    return 0;
}
```

一直以来，超级块都没有写入 `disk`，现在我们将超级块信息写入

```
void finish_fs(char *name) {
    int fd, i;

    // Prepare super block.
    memcpy(disk[1].data, &super, sizeof(super));
```

最后我们将 `disk` 中所有的 `data` 写入一个新创建的文件中，作为磁盘镜像。

```

    // Dump data in `disk` to target image file.
    fd = open(name, O_RDWR | O_CREAT, 0666);
    for (i = 0; i < 1024; ++i) {
#ifdef CONFIG_REVERSE_ENDIAN
        reverse_block(disk + i);
#endif
        ssize_t n = write(fd, disk[i].data, BY2BLK);
        assert(n == BY2BLK);
    }

    // Finish.
    close(fd);
}
```

> `reverse_block` 用于宿主机和操作系统大小端不一致的情况，这里我们不需要考虑，就不看了。

这样我们就实现了磁盘镜像的创建。在阅读 fsformat 工具的代码的同时，我们也了解了磁盘中数据的组织形式以及文件系统的基本概念，这对我们之后了解理解文件系统的代码有很大的帮助。

三、文件系统

（1）文件操作库函数

操作系统需要为用户程序提供一系列库函数来完成文件的相关操作。这些函数我们都知道，有 `open`、`read`、`write`、`close` 等。我们先以 `open` 为例查看一下文件系统操作的基本流程。

`open` 函数和其他提供给用户的库函数一样，位于 user/lib，定义在 file.c 文件中。首先，该函数调用 `fd_alloc` 申请了一个文件描述符

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

    // Step 1: Alloc a new 'Fd' using 'fd_alloc' in fd.c.
    // Hint: return the error code if failed.
    struct Fd *fd;
    /* Exercise 5.9: Your code here. (1/5) */
    try(fd_alloc(&fd));
```

文件描述符 `struct Fd` 定义在 user/include/fd.h 中，保存了文件对应的设备 `fd_dev_id`，文件读写的偏移量 `fd_offset` 和文件读写的模式 `fd_omode`。该结构体不表现文件的物理结构，是在用户侧对文件的抽象。

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

`fd_alloc` 函数的功能就是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。注意通过文件描述符编号到地址我们使用了宏 `INDEX2FD`。该宏定义在 user/include/fd.h 中。通过观察相关宏的定义，我们可以得知每个文件描述符占用空间大小为 `BY2PG`，所有文件描述符位于 `[FILEBASE - PDMAP, FILEBASE)` 的地址空间中。

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

#define INDEX2FD(i) (FDTABLE + (i)*BY2PG)
```

这里我们并没有采用任何数据结构用于表示文件描述符的分配，而是通过查看页目录项和页表项是否有效来得知文件描述符是否被使用。这是因为文件描述符并不是在用户进程中被创建的，而是在**文件系统服务进程**中创建，被共享到用户进程的地址区域中的。因此我们只需要找出一处空闲空间，将文件系统服务进程中的对应文件描述符共享到该位置即可。

```
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) {
```

```
            *fd = (struct Fd *)va;
            return 0;
        }

        if ((vpt[va / BY2PG] & PTE_V) == 0) { // the fd is not used
            *fd = (struct Fd *)va;
            return 0;
        }
    }

    return -E_MAX_OPEN;
}
```

> MOS 采取了微内核设计，因此文件系统的大部分操作并不在内核态中完成，而是交由一个**文件系统服务进程**处理。之后我们会了解这一进程的相关内容。

> `fd_alloc` 中我们先查看页目录项，如果页目录项无效，那么整个页目录项对应的地址空间就都没有进行映射，那么只需要返回 `va` 即可，因为 `va` 就是该目录项的第一个页表中的第一个页表项对应的地址，即第一个没有被使用的文件描述符的地址。

需要注意，我们这里只是返回了可以作为文件描述符的地址，还没有实际的文件描述符数据。之后我们就要获取文件描述符。在 `open` 中调用 `fsipc_open`。该函数会将 `path` 对应的文件以 `mode` 的方式打开，将该文件的文件描述符共享到 `fd` 指针对应的地址处。

该函数定义在 user/lib/fsipc.c 中。所做的工作很简单，就是通过进程间通信向文件系统服务进程发送一条消息，表示自己希望进行的操作，服务进程再返回一条消息，表示操作的结果。这样一来一回，虽然不是函数调用，却产生了函数调用的效果，很是像系统调用。

`fsipc_open` 中，我们将一块缓冲区 `fsipcbuf` 视为 `struct Fsreq_open`，向其中写入了请求打开的文件路径 `req_path` 和打开方式 `req_omode`。并调用 `fsipc` 进行发送。

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

    req = (struct Fsreq_open *)fsipcbuf;

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

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

缓冲区 `fsipcbuf` 是一个页面。因为其大小为 `BY2PG` 字节，又以 `BY2PG` 为基准对齐。

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

`Fsreq_open` 定义在 user/include/fsreq.h 中，类似的还有 `Fsreq_close`、`Fsreq_map` 等等。

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

在调用 `fsipc` 时我们的第一个参数表示请求的类型。这些类型也都定义在 user/include/fsreq.h 中。

```
#define FSREQ_OPEN 1
#define FSREQ_MAP 2
#define FSREQ_SET_SIZE 3
#define FSREQ_CLOSE 4
#define FSREQ_DIRTY 5
#define FSREQ_REMOVE 6
#define FSREQ_SYNC 7
```

`fsipc` 函数就是简单的向服务进程发送消息，并接收服务进程返回的消息。注意这里我们通过 `envs[1].env_id` 获取服务进程的 envid，这说明服务进程必须为第二个进程。

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

（2）文件系统服务进程的初始化

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。我们从这里入手。

首先调用 `serve_init` 对程序进行初始化。

```
int main() {
    user_assert(sizeof(struct File) == BY2FILE);

    debugf("FS is running\n");

    serve_init();
```

在 serve_init 函数中，实际进行初始化的只有 opentab。这是一个 struct Open 类型的数组，用于记录整个操作系统中所有处于打开状态的文件。MAXOPEN 就表示了文件打开的最大数量。

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

// initialize to force into data section
struct Open opentab[MAXOPEN] = {{0, 0, 1}};
```

struct Open 定义在同一个文件中。其中保存了打开的文件 o_file、文件的 id o_fileid、文件打开的方式 o_mode 和文件对应的文件描述符 o_ff。

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

Filefd 是如下所示的结构体。我们在将文件描述符共享到用户进程时，实际上共享的是 Filefd。

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

在 serve_init 中，我们就是对 opentab 进行初始化。为其中每个元素设定 o_fileid 和 o_ff。

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

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

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

地址的分配是从 va = FILEVA 开始的，其中 #define FILEVA 0x60000000。每个 Filefd 分配一页 BY2PG 的大小。因此所有的 Filefd 存储在 [FILEVA, FILEVA + PDMAP) 的地址空间中。

在 main 函数中，之后我们又调用 fs_init 完成文件系统的初始化。

```
    fs_init();
```

该函数定义在 fs/fs.c 中。其中又调用了三个函数 read_super、check_write_block 和 read_bitmap

```
void fs_init(void) {
    read_super();
    check_write_block();
    read_bitmap();
}
```

read_super 中大部分都是检查，只有如下部分值得关注。该部分读取了第一个磁盘块的内容，赋值给全局变量 super。

```
    // Step 1: read super block.
    if ((r = read_block(1, &blk, 0)) < 0) {
        user_panic("cannot read superblock: %e", r);
    }

    super = blk;
```

其中 read_block 用于读取对应磁盘块编号的磁盘块数据到内存中。我们不看前面的检查。首先使用 diskaddr 获取磁盘块编号应该存储到的地址。

```
int read_block(u_int blockno, void **blk, u_int *isnew) {
    // omit...

    // Step 3: transform block number to corresponding virtual address.
    void *va = diskaddr(blockno);
```

diskaddr 函数很简单。这里只是人为规定的地址，我们将 [DISKMAP, DISKMAP+DISKMAX) 的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。因为该缓冲区与内存空间是一一映射的，所以我们也可以得知实验中支持的最大磁盘大小为 DISKMAX = 1GB。

```
void *diskaddr(u_int blockno) {
    /* Exercise 5.6: Your code here. */
    return DISKMAP + blockno * BY2BLK;
}
```

回到 read_block，接下来通过 block_is_mapped 判断编号对应的磁盘块是否已经被映射，如果没有，则我们需要为其分配内存，并将硬盘中的数据读入该内存空间中。最后我们设置 *blk 以返回读取的磁盘块数据对应的内存地址。

```
    if (block_is_mapped(blockno)) { // the block is in memory
        if (isnew) {
            *isnew = 0;
        }
    } else { // the block is not in memory
        if (isnew) {
            *isnew = 1;
        }
        syscall_mem_alloc(0, va, PTE_D);
        ide_read(0, blockno * SECT2BLK, va, SECT2BLK);
    }

    // Step 5: if blk != NULL, assign 'va' to '*blk'.
    if (blk) {
        *blk = va;
    }
    return 0;
}
```

这一部分使用的 block_is_mapped 函数比较简单。和我们在 fd_alloc 中遇到的通过页表来判断是否被使用相同。

```
// Overview:
//  Check if this virtual address is mapped to a block. (check PTE_V bit)
int va_is_mapped(void *va) {
    return (vpd[PDX(va)] & PTE_V) && (vpt[VPN(va)] & PTE_V);
}

// Overview:
//  Check if this disk block is mapped in cache.
//  Returns the virtual address of the cache page if mapped, 0 otherwise.
void *block_is_mapped(u_int blockno) {
    void *va = diskaddr(blockno);
    if (va_is_mapped(va)) {
        return va;
    }
    return NULL;
}
```

如果没有映射，则我们调用 syscall_mem_alloc 为该地址分配一页空间，之后调用 ide_read 函数从磁盘中读取数据。这一部分内容属于磁盘驱动，我们会在下一章讲解。

现在让我们回到 fs_init。check_write_block 函数只是一个插入到基本流程中的测试代码，不需要考虑。我们考虑 read_bitmap。该函数会将管理磁盘块分配的位图读取到内存中。

这里很奇怪的使用了 `super->s_nblocks / BIT2BLK + 1` 来计算作为位图的磁盘块数量。但是对于 `super->s_nblocks` 是 `BIT2BLK` 整倍数的情况，不是会多计算一个磁盘块吗？不懂。（不过 nbitmap 多了 1 似乎也不会产生什么大的影响，因为各个磁盘块数据在内存中的空间并不重合。）之后就是调用 `read_block` 将磁盘上的数据读取到内存缓冲区中，最后设定全局变量 `bitmap` 的值为位图的首地址。

```
void read_bitmap(void) {
    u_int i;
    void *blk = NULL;

    // Step 1: Calculate the number of the bitmap blocks, and read them into
memory.
    u_int nbitmap = super->s_nblocks / BIT2BLK + 1;
    for (i = 0; i < nbitmap; i++) {
        read_block(i + 2, blk, 0);
    }

    bitmap = diskaddr(2);
```

`read_bitmap` 中之后的内容都是一些检查，这里就不详细说明了。

（3）文件系统服务进程的服务

初始化终于完成了，接下来我们调用 `serve` 以开启服务

```
    serve();
```

很容易理解，这样的服务进程就是一个死循环。不断地调用 `ipc_recv` 以接收其他进程发来的请求，根据请求类型的不同分发给不同的处理函数进行处理，并进行回复。`serve` 函数中只需要注意一点，那就是在完成处理后需要进行系统调用 `syscall_mem_unmap` 以取消接收消息时的页面共享，为下一次接收请求做准备。

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

    for (;;) {
        perm = 0;

        req = ipc_recv(&whom, (void *)REQVA, &perm);

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

        switch (req) {
        case FSREQ_OPEN:
            serve_open(whom, (struct Fsreq_open *)REQVA);
```

```
            break;

        // omit...
        }

        syscall_mem_unmap(0, (void *)REQVA);
    }
}
```

这里我们只考虑 FSREQ_OPEN 请求的处理函数 serve_open。在该函数中，我们首先调用 alloc_open 申请一个存储文件打开信息的 struct Open 控制块。需要注意这里的 ipc_send 类似于发生错误时的返回。只不过不知为何此处不和下面调用 file_open 时一样有 return 语句。这或许是一个**错误**，不过既然此处并没有让人写代码，那还是不要擅自改动好了

```
void serve_open(u_int envid, struct Fsreq_open *rq) {
    struct File *f;
    struct Filefd *ff;
    int r;
    struct Open *o;

    // Find a file id.
    if ((r = open_alloc(&o)) < 0) {
        ipc_send(envid, r, 0, 0);
    }
```

open_alloc 函数用于申请一个未使用的 struct Open 元素。这里由于 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（第三次！！！）就可以得知 struct Open 元素是否被使用了。（注意这里不能查看 opentab 中各元素的页表项，因为 opentab 作为数组，占用的空间已经被分配了。）

```
int open_alloc(struct Open **o) {
    int i, r;

    // Find an available open-file table entry
    for (i = 0; i < MAXOPEN; i++) {
        switch (pageref(opentab[i].o_ff)) {
        case 0:
            if ((r = syscall_mem_alloc(0, opentab[i].o_ff, PTE_D | PTE_LIBRARY)) <
0) {
                return r;
            }
        case 1:
            opentab[i].o_fileid += MAXOPEN;
            *o = &opentab[i];
            memset((void *)opentab[i].o_ff, 0, BY2PG);
            return (*o)->o_fileid;
        }
    }
```

```
    return -E_MAX_OPEN;
}
```

通过 pageref 我们可以得知某一页的引用数量。该函数定义在 user/lib/pageref.c 中，不需要多言。

```
int pageref(void *v) {
    u_int pte;

    if (!(vpd[PDX(v)] & PTE_V)) {
        return 0;
    }

    pte = vpt[VPN(v)];

    if (!(pte & PTE_V)) {
        return 0;
    }

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

我必须解释一下 open_alloc 中 switch 的使用。首先要再次明确 pageref 返回的是某一页的引用数量。那么除了 0、1 以外，还可能有 2、3 等等，即物理页在不同进程间共享的情况。在我们的文件系统中，是会出现将 Filefd 共享到用户进程的情况，这时因为 switch 的 case 中只有 1、2，因此便会跳过这次循环。这样我们就将正在使用的文件排除在外了。

我们知道在最开始，所有的 Filefd 都没有被访问过，他们的引用数量为 0。只有当使用过之后，引用数量才大于 0。那么 case 1 表示的情况就很明显了，那就是曾经被使用过，但现在不被任何用户进程使用的文件，只有服务进程还保存着引用。这种情况的 struct Open 就没有被使用，因此可以被申请。但是之前使用时的 o_fileid 这次就不能使用了，需要更新 opentab[i].o_fileid += MAXOPEN，同时将对应的文件描述符 o_ff 的内容清零。

最后是还没有被访问过的情况，这种情况下我们先要使用系统调用 syscall_mem_alloc 申请一个物理页。注意申请时我们使用的权限位 PTE_D | PTE_LIBRARY。PTE_LIBRARY 表示该页面可以被共享。之后呢？我们还需要和引用数量为 1 的情况一样，将 o_ff 对应的空间清零，返回 o_fileid 和 opentab[i]。这里采用了一个巧妙的方法，在 case 0 和 case 1 之间没有使用 break 分隔，直接让 case 0 的执行穿透到了 case 1 中。

现在让我们回到 serve_open。接下来调用 file_open 来打开文件。

```
    // Open the file.
    if ((r = file_open(rq->req_path, &f)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
```

file_open 定义在 fs/fs.c 中，只是调用了 walk_path 而已。

```
int file_open(char *path, struct File **file) {
    return walk_path(path, 0, file, 0);
}
```

`walk_path` 函数位于同一个文件中，主要内容就是解析路径，根据路径不断找到目录下的文件，找到最后得到的就是表示路径对应的文件的 `struct File`。

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

    // start at the root.
    path = skip_slash(path);
    file = &super->s_root;
    dir = 0;
    name[0] = 0;

    if (pdir) {
        *pdir = 0;
    }

    *pfile = 0;

    // find the target file by name recursively.
    while (*path != '\0') {
        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;
}
```

该函数大部分工作都用于完成路径解析和异常处理，不用解释相信各位也能理解。唯一需要注意的是 `dir_lookup` 函数。该函数用于找到指定目录下的指定名字的文件。该函数也位于同一个文件中，也是我们需要填写代码的函数。

该函数本身很简单，与 tools/fsformat.c 中的 `create_file` 类似。都是获取文件的所有磁盘块，遍历其中所有的 `struct File`。只不过这里需要返回指定名字的文件对应的 `struct File`。

```
int dir_lookup(struct File *dir, char *name, struct File **file) {
    int r;
    // Step 1: Calculate the number of blocks in 'dir' via its size.
    u_int nblock;
    /* Exercise 5.8: Your code here. (1/3) */
    nblock = dir->f_size / BY2BLK;

    // Step 2: Iterate through all blocks in the directory.
    for (int i = 0; i < nblock; i++) {
        // Read the i'th block of 'dir' and get its address in 'blk' using
'file_get_block'.
        void *blk;
        /* Exercise 5.8: Your code here. (2/3) */
        try(file_get_block(dir, i, &blk));

        struct File *files = (struct File *)blk;

        // Find the target among all 'File's in this block.
        for (struct File *f = files; f < files + FILE2BLK; ++f) {
            /* Exercise 5.8: Your code here. (3/3) */
            if (strcmp(name, f->f_name) == 0) {
                *file = f;
                f->f_dir = dir;
```

```
                return 0;
            }
        }
    }

    return -E_NOT_FOUND;
}
```

唯一需要注意的是 `file_get_block` 函数。该函数用于获取文件中第几个磁盘块。其中首先调用 `file_map_block` 获取了文件中使用的第几个磁盘块对应的磁盘块编号（请注意这两者的区别，一个是相对于文件中其他部分的编号，另一个是相对于磁盘来说的编号）。

```
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;
    }
```

在 `file_map_block` 中首先调用 `file_block_walk` 获取对应的磁盘块编号

```
int file_map_block(struct File *f, u_int filebno, u_int *diskbno, u_int alloc) {
    int r;
    uint32_t *ptr;

    // Step 1: find the pointer for the target block.
    if ((r = file_block_walk(f, filebno, &ptr, alloc)) < 0) {
        return r;
    }
```

`file_block_walk` 中用到了之前着重解释的 `f_direct` 和 `f_indirect`。容易发现此函数的与 fsformat 中的 `save_block_link` 函数结构类似。此函数我们已经在之前说明过了。需要注意的是这里当 `f_indirect` 还未申请时，我们使用了 `alloc_block` 来申请一个新的磁盘块，并使用 `read_block` 将该磁盘块数据读入内存中。

```
int file_block_walk(struct File *f, u_int filebno, uint32_t **ppdiskbno, u_int
alloc) {
    int r;
    uint32_t *ptr;
    uint32_t *blk;

    if (filebno < NDIRECT) {
        // Step 1: if the target block is corresponded to a direct pointer, just
return the
        // disk block number.
```

```
        ptr = &f->f_direct[filebno];
    } else if (filebno < NINDIRECT) {
        // Step 2: if the target block is corresponded to the indirect block, but
there's no
        //  indirect block and `alloc` is set, create the indirect block.
        if (f->f_indirect == 0) {
            if (alloc == 0) {
                return -E_NOT_FOUND;
            }

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

        // Step 3: read the new indirect block to memory.
        if ((r = read_block(f->f_indirect, (void **)&blk, 0)) < 0) {
            return r;
        }
        ptr = blk + filebno;
    } else {
        return -E_INVAL;
    }

    // Step 4: store the result into *ppdiskbno, and return 0.
    *ppdiskbno = ptr;
    return 0;
}
```

`read_block` 我们之前已经提及了，现在我们就考察一下 `alloc_block`。该函数首先调用 `alloc_block_num` 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

```
int alloc_block_num(void) {
    int blockno;
    // walk through this bitmap, find a free one and mark it as used, then sync
    // this block to IDE disk (using `write_block`) from memory.
    for (blockno = 3; blockno < super->s_nblocks; blockno++) {
        if (bitmap[blockno / 32] & (1 << (blockno % 32))) { // the block is free
            bitmap[blockno / 32] &= ~(1 << (blockno % 32));
            write_block(blockno / BIT2BLK + 2); // write to disk.
            return blockno;
        }
    }
    // no free blocks.
    return -E_NO_DISK;
}

int alloc_block(void) {
    int r, bno;
    // Step 1: find a free block.
```

```
    if ((r = alloc_block_num()) < 0) { // failed.
        return r;
    }
    bno = r;
```

之后 alloc_block 函数调用 map_block 将获取的编号所对应的空闲磁盘块从磁盘中读入内存。

```
    // Step 2: map this block into memory.
    if ((r = map_block(bno)) < 0) {
        free_block(bno);
        return r;
    }

    // Step 3: return block number.
    return bno;
}
```

map_block 函数申请一个页面用于存储磁盘块内容，类似的含有 unmap_block 函数。这两个函数较为简单，唯一需要注意的是 unmap_block 会将内存中对磁盘块的修改写回磁盘。

```
int map_block(u_int blockno) {
    // Step 1: If the block is already mapped in cache, return 0.
    // Hint: Use 'block_is_mapped'.
    /* Exercise 5.7: Your code here. (1/5) */
    if (block_is_mapped(blockno)) {
        return 0;
    }

    // Step 2: Alloc a page in permission 'PTE_D' via syscall.
    // Hint: Use 'diskaddr' for the virtual address.
    /* Exercise 5.7: Your code here. (2/5) */
    try(syscall_mem_alloc(env->env_id, diskaddr(blockno), PTE_D));
}

void unmap_block(u_int blockno) {
    // Step 1: Get the mapped address of the cache page of this block using
'block_is_mapped'.
    void *va;
    /* Exercise 5.7: Your code here. (3/5) */
    va = block_is_mapped(blockno);

    // Step 2: If this block is used (not free) and dirty in cache, write it back
to the disk
    // first.
    /* Exercise 5.7: Your code here. (4/5) */
    if (!block_is_free(blockno) && block_is_dirty(blockno)) {
        write_block(blockno);
    }
```

```
    // Step 3: Unmap the virtual address via syscall.
    /* Exercise 5.7: Your code here. (5/5) */
    syscall_mem_unmap(env->env_id, diskaddr(blockno));

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

最后 free_block 则是重新将位图对应位置置 1，表示空闲。

```
void free_block(u_int blockno) {
    // You can refer to the function 'block_is_free' above.
    // Step 1: If 'blockno' is invalid (0 or >= the number of blocks in 'super'),
return.
    /* Exercise 5.4: Your code here. (1/2) */
    if (super == 0 || blockno >= super->s_nblocks) {
        return;
    }

    // Step 2: Set the flag bit of 'blockno' in 'bitmap'.
    // Hint: Use bit operations to update the bitmap, such as b[n / W] |= 1 << (n
% W).
    /* Exercise 5.4: Your code here. (2/2) */
    bitmap[blockno / 32] |= 1 << (blockno % 32);
}
```

我在看这一部分的时候有点想要吐槽一下，为什么 map_block 函数和 read_block 函数如此相似？map_block 分明完全是 read_block 的弱化版。后来回看 file_block_walk 的时候就明白原因了。在 file_block_walk 中我们同时使用了 alloc_block（其中用到 read_block） 和 read_block。其中 alloc_block 只是申请了一个磁盘块，但因为是新申请，所以对应地址空间中的数据没有用处，并不从磁盘中读取数据，只需要申请对应地址的物理页即可。而 read_block 则进行了数据的读取。如在 file_block_walk 中需要读取间接磁盘块中的数据来确定。

经过了这么艰辛的历程，我们重新回到 file_map_block。文件的第几个磁盘块对应的磁盘块编号现在已经被存储在了 *ptr。这里还考虑了未找到时再调用 alloc_block 申请一个磁盘块的情况。

```
    // Step 2: if the block not exists, and create is set, alloc one.
    if (*ptr == 0) {
        if (alloc == 0) {
            return -E_NOT_FOUND;
        }

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

最后将文件的第几个磁盘块对应的磁盘块编号传给 *diskbno，这样就找到了对应的磁盘块编号。

```
    // Step 3: set the pointer to the block in *diskbno and return 0.
    *diskbno = *ptr;
    return 0;
}
```

还记得我们是从哪里调用的吗？我们返回到 file_get_block，现在我们已经找到了文件中第几个磁盘块对应的磁盘块编号，最后只需要调用 read_block 将该磁盘块的内容从磁盘中读取到内存即可。

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

之后，我们的 dir_lookup 函数就可以遍历目录文件下所有的文件，找到和目标文件名相同的文件了。而 dir_lookup 作为 walk_path 的重要组成部分，使 walk_path 完成了根据路径获取对应文件的功能。file_open 函数调用 walk_path 之后返回。我们终于又回到了 serve_open 函数。

serve_open 接下来的内容就比较简单了，只是将 file_open 返回的 struct File 结构体设置到 struct Open 结构体，表示新打开的文件为该文件，接着设置一系列字段的值。最后调用 ipc_send 返回，将文件描述符 o->o_ff 与用户进程共享。

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

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

    ipc_send(envid, 0, o->o_ff, PTE_D | PTE_LIBRARY);
}
```

只需要注意 ff->f_fd.fd_dev_id = devfile.dev_id; 这一句，我们设置文件描述符对应的设备为 devfile。该变量定义在 user/lib/file.c 中

```
struct Dev devfile = {
    .dev_id = 'f',
    .dev_name = "file",
```

```
    .dev_read = file_read,
    .dev_write = file_write,
    .dev_close = file_close,
    .dev_stat = file_stat,
};
```

其中 `struct Dev` 定义在 user/include/fd.h 中

```
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);
};
```

这样看就很容易理解了。这实际上通过结构体实现了类似抽象类的功能。

（4）open 的后续

不要忘了，我们的 `open` 函数还没有结束呢。接着上一节，获取到的文件描述符与用户进程共享，那么共享到哪里了呢？如果你还记得 `fd_alloc` 函数，那么应该知道共享到了 `struct Fd *fd` 所指向的地址处。虽然我们获得了服务进程共享给用户进程的文件描述符，可文件的内容还没有被一同共享过来。我们还需要使用 `fsipc_map` 进行映射。

在此之前我们先做准备工作。我们通过 `fd2data` 获取文件内容应该映射到的地址

```
    // Step 3: Set 'va' to the address of the page where the 'fd''s data is
cached, using
    // 'fd2data'. Set 'size' and 'fileid' correctly with the value in 'fd' as a
'Filefd'.
    char *va;
    struct Filefd *ffd;
    u_int size, fileid;
    /* Exercise 5.9: Your code here. (3/5) */
    va = fd2data(fd);
```

由定义可知，该函数为不同的文件描述符提供不同的地址用于映射。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。

```
// user/lib/fd.c
void *fd2data(struct Fd *fd) {
    return (void *)INDEX2DATA(fd2num(fd));
}
```

```
int fd2num(struct Fd *fd) {
    return ((u_int)fd - FDTABLE) / BY2PG;
}

// user/include/fd.h
#define FILEBASE 0x60000000

#define INDEX2DATA(i) (FILEBASE + (i)*PDMAP)
```

接着我们将文件所有的内容都从磁盘中映射到内存。使用的函数为 `fsipc_map`。映射的过程和得到文件描述符的过程类似，甚至更简单，就不详述了。

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

    // Step 4: Alloc pages and map the file content using 'fsipc_map'.
    for (int i = 0; i < size; i += BY2PG) {
        /* Exercise 5.9: Your code here. (4/5) */
        try(fsipc_map(fileid, i, va));
    }
```

在最后，使用 `fd2num` 方法获取文件描述符在文件描述符 “数组” 中的索引

```
    // Step 5: Return the number of file descriptor using 'fd2num'.
    /* Exercise 5.9: Your code here. (5/5) */
    return fd2num(fd);
}
```

这样，`open` 函数就终于完成了。

（5）read 的实现

文件系统的最后一部分，让我们再举 `read` 做一个例子。该函数位于 user/lib/fd.c 中。

`read` 函数给了文件描述符序号作为参数，我们首先要根据该序号找到文件描述符，并根据文件描述符中的设备序号 `fd_dev_id` 找到对应的设备。这两个操作分别通过 `fd_lookup` 和 `dev_lookup` 实现。

```
int read(int fdnum, void *buf, u_int n) {
    int r;

    // Similar to the 'write' function below.
    // Step 1: Get 'fd' and 'dev' using 'fd_lookup' and 'dev_lookup'.
    struct Dev *dev;
    struct Fd *fd;
    /* Exercise 5.10: Your code here. (1/4) */
    if ((r = fd_lookup(fdnum, &fd)) < 0 || (r = dev_lookup(fd->fd_dev_id, &dev)) <
```

```
0) {
        return r;
    }
```

因为文件描述符存储在内存中的连续空间中，所以 fd_lookup 函数只是根据序号找到对应的文件描述符，并且判断文件描述符是否被使用而已，这一点函数中又一次使用页表判断。

```
int fd_lookup(int fdnum, struct Fd **fd) {
    u_int va;

    if (fdnum >= MAXFD) {
        return -E_INVAL;
    }

    va = INDEX2FD(fdnum);

    if ((vpt[va / BY2PG] & PTE_V) != 0) { // the fd is used
        *fd = (struct Fd *)va;
        return 0;
    }

    return -E_INVAL;
}
```

对于 dev_lookup，首先我们应该了解到所有的设备都被存储到了全局变量 devtab 中。（这里出现了我们之前见到过的 devfile。devcons 类似。）

```
static struct Dev *devtab[] = {&devfile, &devcons,
#if !defined(LAB) || LAB >= 6
                   &devpipe,
#endif
                   0};
```

那么在 dev_lookup 中，我们就只是遍历该数组，找到和传入的参数 dev_id 相同的设备而已。

```
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;
}
```

回到 read，接下来我们判断文件的打开方式是否是只写，如果是那么我们就不能够进行读取，应该返回异常。

```
    // Step 2: Check the open mode in 'fd'.
    // Return -E_INVAL if the file is opened for writing only (O_WRONLY).
    /* Exercise 5.10: Your code here. (2/4) */
    if ((fd->fd_omode & O_ACCMODE) == O_WRONLY) {
        return -E_INVAL;
    }
```

接着我们调用设备对应的 dev_read 函数，完成数据的读取。

```
    // Step 3: Read from 'dev' into 'buf' at the seek position (offset in 'fd').
    /* Exercise 5.10: Your code here. (3/4) */
    r = dev->dev_read(fd, buf, n, fd->fd_offset);
```

根据之前我们展示的 devfile 的定义，普通文件的读取函数为 file_read。该函数位于 user/lib/file.c 中，只是简单地读取被映射到内存中的文件内容而已。（还记得 fsipc_map 吗？）

```
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;

    // Avoid reading past the end of file.
    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;
}
```

再次回到 read，我们读取完了内容，现在我们要更新文件的指针 fd_offset。在调用读取函数的时候，我们使用 fd_offset 确定了读取的位置（dev_read(fd, buf, n, fd->fd_offset)）。那么下一次读取时，就应该从还未被读取的地方读取了。更新完成后，我们返回读到的数据的字节数。这样 read 函数也完成了。

```
    // Step 4: Update the offset in 'fd' if the read is successful.
    /* Hint: DO NOT add a null terminator to the end of the buffer!
     *  A character buffer is not a C string. Only the memory within [buf, buf+n)
```

```
is safe to
     *  use. */
    /* Exercise 5.10: Your code here. (4/4) */
    if (r > 0) {
        fd->fd_offset += r;
    }

    return r;
}
```

四、磁盘驱动

（1）设备读写系统调用

我们在前面的许多 Lab 中都见到了与外部设备进行交互的代码。我们只需要在对应的物理地址位置写入或读取某些数值，就可以完成与设备的信息传递。现在我们要规范化这一过程，让用户程序也能够实现与设备的直接交互。也就是说，要实现设备读写的系统调用。

这一部分十分简单。根据指导书和注释我们可以得知，只需要完成对三个设备的读写即可。这三个设备是终端、磁盘和时钟。

```
/*
 *  All valid device and their physical address ranges:
 *  * ---------------------------------*
 *  |   device   | start addr | length |
 *  * -----------+------------+--------*
 *  |  console   | 0x10000000 | 0x20   | (dev_cons.h)
 *  |  IDE disk  | 0x13000000 | 0x4200 | (dev_disk.h)
 *  |    rtc     | 0x15000000 | 0x200  | (dev_rtc.h)
 *  * ---------------------------------*
 */
```

在内核中，我们要完成系统调用的实现。这里我们要判断内存的虚拟地址是否处于用户空间以及设备的物理地址是否处于那三个设备的范围内。如果所有检查都合法，则调用 `memcpy` 从内存向设备写入或从设备向内存读取即可，系统调用函数依旧在 kern/syscall_all.c 中实现。

```
int sys_write_dev(u_int va, u_int pa, u_int len) {
    /* Exercise 5.1: Your code here. (1/2) */
    if (is_illegal_va_range(va, len)) {
        return -E_INVAL;
    }

    if ((0x10000000 <= pa && pa + len <= 0x10000020) ||
        (0x13000000 <= pa && pa + len <= 0x13004200) ||
        (0x15000000 <= pa && pa + len <= 0x15000200)) {
        memcpy((void *)(KSEG1 | pa), (void *)va, len);
        return 0;
    }
```

```
    return -E_INVAL;
}

int sys_read_dev(u_int va, u_int pa, u_int len) {
    /* Exercise 5.1: Your code here. (2/2) */
    if (is_illegal_va_range(va, len)) {
        return -E_INVAL;
    }

    if ((0x10000000 <= pa && pa + len <= 0x10000020) ||
        (0x13000000 <= pa && pa + len <= 0x13004200) ||
        (0x15000000 <= pa && pa + len <= 0x15000200)) {
        memcpy((void *)va, (void *)(KSEG1 | pa), len);
        return 0;
    }

    return -E_INVAL;
}
```

另外不要忘记在用户库函数中实现接口，用户的系统调用接口同样还在 user/lib/syscall_lib.c

```
int syscall_write_dev(void *va, u_int dev, u_int len) {
    /* Exercise 5.2: Your code here. (1/2) */
    return msyscall(SYS_write_dev, va, dev, len);
}

int syscall_read_dev(void *va, u_int dev, u_int len) {
    /* Exercise 5.2: Your code here. (2/2) */
    return msyscall(SYS_read_dev, va, dev, len);
}
```

（2）IDE 磁盘读写

最后我们需要实现磁盘的读写操作。在上一章中我们就遇到了 `ide_read` 函数。该函数通过调用系统操作，实现了从磁盘中读取数据到内存中。类似的还有 `ide_write` 函数。这两个函数都是磁盘驱动。

在指导书中给出了操作 IDE 磁盘可能用到的地址偏移。我们只需要读写这些地址即可。

偏移	效果	数据位宽
0x0000	写：设置下一次读写操作时的磁盘镜像偏移的字节数	4 字节
0x0008	写：设置高 32 位的偏移的字节数	4 字节
0x0010	写：设置下一次读写操作的磁盘编号	4 字节
0x0020	写：开始一次读写操作（写 0 表示读操作，1 表示写操作）	4 字节
0x0030	读：获取上一次操作的状态返回值（读 0 表示失败，非 0 则表示成功）	4 字节

偏移	效果	数据位宽
0x4000-0x41ff	读/写：512 字节的读写缓存	/

`ide_read` 和 `ide_write` 定义在 fs/ide.c 中。这两个函数都需要我们自己实现

```
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs) {
    u_int begin = secno * BY2SECT;
    u_int end = begin + nsecs * BY2SECT;

    for (u_int off = 0; begin + off < end; off += BY2SECT) {
        uint32_t temp = diskno;
        /* Exercise 5.3: Your code here. (1/2) */
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_ID, 4);
        temp = begin + off;
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_OFFSET, 4);
        temp = DEV_DISK_OPERATION_READ;
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_START_OPERATION, 4);
        syscall_read_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_STATUS, 4);
        if (temp == 0) {
            panic_on("fail to read from disk");
        }
        syscall_read_dev(dst + off, DEV_DISK_ADDRESS | DEV_DISK_BUFFER, BY2SECT);
    }
}

void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs) {
    u_int begin = secno * BY2SECT;
    u_int end = begin + nsecs * BY2SECT;

    for (u_int off = 0; begin + off < end; off += BY2SECT) {
        uint32_t temp = diskno;
        /* Exercise 5.3: Your code here. (2/2) */
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_ID, 4);
        temp = begin + off;
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_OFFSET, 4);
        syscall_write_dev(src + off, DEV_DISK_ADDRESS | DEV_DISK_BUFFER, BY2SECT);
        temp = DEV_DISK_OPERATION_WRITE;
        syscall_write_dev(&temp, DEV_DISK_ADDRESS | DEV_DISK_START_OPERATION, 4);
        syscall_read_dev(&temp, DEV_DISK_ADDRESS| DEV_DISK_STATUS, 4);
        if (temp == 0) {
            panic_on("fail to write disk");
        }
    }
}
```

这里的代码看上去复杂，实际上只实现了简单的步骤，比如对于 `ide_read`，只不过实现了 1. 设定要读的磁盘编号；2. 设定要读取的地址；3. 开始读取；4. 获取读取后状态（返回值），如果读取失败则 panic；5. 将缓冲区中的内容读到内存中。

这样 Lab5 就完成了。