作业设计文档

1. 设计内容

本次作业设计并实现了一个外存模拟程序。 作业中在内存中申请了线性数组，模拟了4095个盘块，每个盘块有512byte，共占空间2MB。本次作业中实现了如下命令：

命令	功能	完成情况
blockinfo	查看盘块信息	已完成
fileinfo	查看文件信息	已完成
pwd	查看当前工作目录	已完成
mkdir	创建文件夹	已完成
rmdir	删除文件夹	已完成
cd	切换文件夹	已完成
ls	列出文件	已完成
create	创建文件	已完成
delete	删除文件	已完成
mv	移动文件	未完成

2. 小组成员即分工情况

???

3. 设计环境

本次作业的开发环境如下

key	value
CPU	Ryzen 5600x
内存	DDR4 64G
硬盘	nvme 250G
系统	windows 11
编辑器	DEV CPP
编译器	MinGW GCC
语言标准	c99

本次作业的理想应用环境是在边缘计算设备的最小文件系统。 意在与解决无法运行大型操作系统的低性能设备， 还需要持久化存储并索引较大量的配置文件的专用需求。

4. 设计过程

- 盘块的实现

每个盘块的数据结构如下

```
struct Block {
    int next;                           // 下个盘块
    char data[DATA_SIZE];               // 盘块数据
};
```

其中BLOCK_SIZE = 512是预设值，可以自行更改，DATA_SIZE 是块大小减去块开头的第一个指针的大小

- 盘块的存储位置

由于本次作业是模拟环境，因此盘块将存储在文件系统结构中。

```
struct MyFileSystem {
    struct Block blocks[BLOCK_NUM];
    int free_blocks;
    int current_dir;
    int free_blocks_size;
};
```

BLOCK_NUM=521是预设值，可以自行更改。

- 盘块的初始化过程

文件系统和盘块将由此函数进行初始化

```
// 文件系统初始化
struct MyFileSystem * filesystem_init() {

    // 申请内存
    struct MyFileSystem * fs = malloc(sizeof(struct MyFileSystem));

    // 将所有盘块整理成静态栈
    for (int i = 0; i < BLOCK_NUM; i++) {
        fs->blocks[i].next = i+1;                           // 每个盘块都指向下一盘
块
    }
    fs->free_blocks = 0;                                    // 空闲盘块栈顶
    fs->free_blocks_size = BLOCK_NUM;                       // 空闲盘块数量
    fs->current_dir = 0;                                    // 当前目录
```

```
    // 创建根目录
    mkdir(fs, "/");
    return fs;
}
```

首先，他将申请足够的内存，据计算，约2MB。
其次，随后，它将所有盘块的next指向下一盘块，并将free_blocks指向第一个盘块。
这是因为，我选择使用静态栈数据结构管理空闲盘块。
这样做的好处是在优先使用外圈磁道，且先放回先使用，集中文件在磁盘外圈。读取速度更快，

- 盘块的申请和释放

空闲盘块采用静态栈的方式组织。
盘块的申请，即静态栈出栈。缓存栈顶，将栈顶下标指向下一盘块，弹出栈顶并返回。
盘块的释放，将盘块的next指向栈顶，并将栈顶下标指向当前盘块，即完成释放。
此两函数还递增和递减了计数器，可以方便的知道当前剩余盘块数量，你可以使用blockinfo命令查看盘块状态。

```
// 申请一个块
int block_pop(struct MyFileSystem * fs) {
    int blkid = fs->free_blocks;                              // 即将弹出的block_id
    if (blkid != BLOCK_NULL) {                                // 若空 返回BLOCK_NULL
        fs->free_blocks = fs->blocks[fs->free_blocks].next; // 出栈即将返回的盘块
        fs->blocks[blkid].next = BLOCK_NULL;                  // 清理即将返回的盘块
        -- fs->free_blocks_size;                              // 计数
    }
    return blkid;                                             // 返回
}

// 释放一个块
void block_push(struct MyFileSystem * fs, int blkid) {
    fs->blocks[blkid].next = fs->free_blocks;                 // 入栈
    fs->free_blocks = blkid;                                  // 入栈
    ++ fs->free_blocks_size;                                  // 计数
}
```

- 文件/目录数据结构

```
// 节点类型
enum EntryType {
    ENTRY_FILE,                         // 文件
    ENTRY_DIRECTORY                     // 目录
};

// 节点
struct Entry {
    int block_id;
    char name[MAX_NAME_LENGTH];         // 文件名 / 目录名
```

```
    enum EntryType type;                // 类型 文件 / 目录
    off_t size;                         // 文件大小 / 目录文件数量
    time_t creation_time;               // 创建日期
    int first_block;
    int last_block;
};

// 目录
struct Directory {
    struct Entry entry;
    int parent;                         // 父目录所在盘块id
    int child_head;                     // 首个子节点所在盘块id
    int child_tail;                     // 尾个子节点所在盘块id
};
```

- 文件/目录与盘块的关系

创建目录/文件时，申请盘块存储元数据。
创建文件时还需申请足额的盘块存储文件数据。 删除目录/文件时 归还盘块 使用如下转换函数 直接将申请到的盘块的数据段映射为目录/文件元数据结构

```
struct Block * to_block(struct MyFileSystem * fs, int block_id) {
    return (struct Block*)&fs->blocks[block_id];
}

struct Directory * to_dirctory(struct MyFileSystem * fs, int block_id) {
    return (struct Directory *)fs->blocks[block_id].data;
}
```

- 子节点链表的遍历方法

节点数据中含有 parent, child_head, child_tail三个值。
其分别代表父节点所在盘块，子节点链表的起始盘块，子节点链表的结尾盘块。
子节点链表的next指针借用盘块指针实现。
以遍历 current dir为例

```
// 获取当前目录元数据
struct Directory * current_dir = to_dirctory(fs, fs->current_dir);

// 使用变种的for循环遍历子节点链表
for (
    int block = current_dir->child_head;
    block != BLOCK_NULL;
    block = to_block(fs, block)->next
) {
    // 子节点元数据
    struct Directory * sub_dir = to_dirctory(fs, block);
```

```
    // do something...
}
```

- 创建子目录

  使用了mkdir函数实现了创建子目录的功能。
  其使用filesystem中的currentdir索引到当前路径所在盘块 解析此盘块中存储的元数据，在子节点链表下追加一个文件夹即可。 为了简洁，根目录也使用了mkdir创建，在mkdir中加入了if else 来进行逻辑区分。 如果新目录名字是"/" 则视为创建根目录。

- 删除子目录

  同上的方式获取当前目录元数据。
  从子节点链表中查询同名文件并移除。

- 进入子目录

  若子目录存在，将fs.current_dir赋值为子目录所在盘块即可。

- 输出当前路径

  输出当前路径由此两个函数组合而成
  递归查找父目录 直到根目录停止递归
  随后回溯输出目录 直到当前路径

```
// 输出当前路径 辅助递归函数
void recursion_pwd(
    struct MyFileSystem * fs,
    struct Directory * current_dir
) {
    if (current_dir->parent != BLOCK_NULL) {
        recursion_pwd(fs, to_dirctory(fs, current_dir->parent));
        printf("/");
    }
    if (current_dir->parent == BLOCK_NULL) {
        return;
    }

    printf("%s", current_dir->entry.name);
}

// 输出当前路径
void pwd(struct MyFileSystem * fs) {
    recursion_pwd(fs, to_dirctory(fs, fs->current_dir));
    printf("/");
}
```

- 创建文件

创建文件 可部分沿用创建目录的逻辑
还需要检查盘块是否足够
并且循环申请盘块组成数据链表
并将数据链表的表头表尾写入文件元数据中

- 删除文件

  删除文件 可部分沿用删除目录的逻辑
  还需循环删除并归还数据盘块

- 移动文件

  由于时间紧张 暂未实现

2. 遇到的问题

层因文件/目录元数据设计不当，将子节点和兄弟节点置入了统一条盘块链表中。
导致无法组织线性数据，且存储文件数据时会抹除子节点

3. 解决办法

新的设计将使用两条链表分别保存兄弟节点和子节点，并存储了当前节点和父节点，避免了问题的发生。

5 设计成果

1. 使用情况

2. 运行情况

3. 运行截图

6. 设计总结

本次课题实现了虚拟磁盘管理器，基本实现了大部分功能。
在平常的编程中，size_t即是最大的值，因为其不会超出内存上限。
但磁盘动辄高达TB，将使用新的单位off_t来计算文件大小和盘块号。
这让我对单位临界值的考量有了新的认识。

在自己的程序中模拟一个文件管理系统，当其成功运行时，获得了莫大的成就感。

不足

> 由于时间紧张，暂未实现移动文件，实属遗憾。
> 暂未实现B树等更高级的索引，在同一路径下文件数量过万后，可能会产生索引效率问题。
> 但好在实现了树状文件结构，在多数情况下依然可用。

改善方案

后续将持续改进，增加对文件移动的支持。
可能的实现思路： 将文件的元数据移出某目录的子节点链表，置入其他目录的子节点链表。
后续将会加入B树等索引功能，提升索引性能。