Project 4a: File Defragmentation

10185102223 汪杰

格式假设

为了保证代码正确运行，需要对硬盘格式做出一定假设：

- **Superblock**和**Inode**的结构与给定结构体相同，见 defrag.h
- 硬盘中按次序分别是：boot block、superblock、inode region、data region，且中间没有空隙。
- 块大小是2的幂次
- 假设不存在swap region，即data region延续到硬盘末尾。
    - 在本实验中，swap region是假的，如果查看super block中的swap_offset，会发现这个偏移量指向硬盘外。因此，本程序不考虑这个区域。
    - 带来的后果将会是：若swap region真实存在，那么本程序执行后，空闲块会蔓延到swap region内。不过这个bug是容易修复的。

除此之外，不做更多的假设，以获得尽可能高的普适性与鲁棒性。例如：

- 对每个区域的大小不做限制，可以是任意的。
    - 不限制boot block和super block的大小，且不必与块大小一致（虽然通常是一个块）
        - 在 defrag.h 中定义其大小
    - 不假设块大小为512B，不限制inode region、data region的大小（只要保持顺序）
        - 这些值都将从super block中获取。
- defrag.h 中的一些宏定义是可以改变的。
    - **Inode**结构体中的两个数组大小 `N_DBLOCKS` 和 `N_IBLOCKS` 可变。
    - 未使用块的填充值定义在 `EMPTY_DATA` ，通常可以是0或-1

注意到，假设中有一条要求：块大小是2的幂次。

- 通常，根据下标定位块的位置时需要用下标乘以块的大小。乘法是较慢的。在块的大小是2的幂次这一前提下，就可以用位运算加速。
- 本程序假定块的大小是2的幂次，并用左移和右移运算代替乘法和除法。
- 但是，这个优化的作用不是太大。因为性能瓶颈是复制文件，而非定位块的位置。只能说，聊胜于无。

执行流程

1. 尝试打开输入文件
2. 复制 boot block
3. 将 super block 读入到内存，并写入输出文件
    - 这里的复制其实是无效的，因为之后还要修改 `free_iblock` 。super block会在最后更新，并再次写入这个位置。
    - 这里写入的目的是扩充输出文件大小。因为此时，输出文件还只有 512B （只包含boot block）

4. 将 inode region 读入到内存，并写入输出文件
    - **Inode** 中的各种索引值还会修改，同理，暂时写入
5. 遍历 inode ，依次复制每个文件，并按需建立对应的间接索引块。索引信息写入内存中的inode中。
6. 填充剩余空闲块，格式化为顺序排列的链表。把链表头（第一个空闲块）的位置写入 super block。
7. 重写super block 和 inode region。
8. 清理已分配内存

内存管理

硬盘文件（通常，虽然测试文件并不是）很大，没有办法将整个硬盘读入内存。

在硬盘碎片整理的过程中，某些部分会高频读写，而某些部分只会读写一次。对于前者，可以全部读取到内存中；而对于后者，只要一次读取/写入一个块，无需整体读取。具体来说：

- super block的信息会被高频地读。每次获取块大小，获取数据区偏移，都要读取super block。
- inode region的内容会被高频读写。获取文件大小及其在硬盘中的位置时要读，而修改为新的位置时要写。
    - 虽然对于一个文件（一个inode），只要读写一次，但是一个块中包含多个inode，加之inode在块中是不对齐的（存在跨块的inode），所以事先缓存是更好的。
- 文件的数据只会被读写（复制）一次，且已经按块分隔好。

在所有读写都完成后，再将内存中的 super block 和 inode region写入硬盘中。从而降低外存的读写次数。

除了上面提到的内存占用之外，在写间接索引块时也需要分配一些内存，而不是直接写入外存：

- 一个间接索引块包含 $512 \div 4 = 128$ 个块下标。如果每个下标都分别写入外存，那么开销是巨大的。
- 所以先把这些下标缓存在内存中，以块为单位整体写入硬盘。
- 这一部分需要分配的内存数量取决于间接索引的最大层数。只要为每一层分配一块内存即可。
    - 不需要为每个间接索引块都分配一块内存。因为某一块间接索引块建立完后，就可以直接写入硬盘
    - 不能少于间接索引的层数，因为（本实验中）建立间接索引的过程是递归的。建立3级间接索引时，会同时存在2级和1级的间接索引。

总结下来，本程序在执行过程中需要占用的额外内存空间为8块（以测试文件为例），即 $8 \times 512\mathrm{B}$：

- super block：1块
- inode region：4块
- 直接索引（文件数据块）：1块
- 间接索引：3块

复制文件

碎片整理无需刻意控制输出的位置。只要读文件是按照顺序的，那么直接输出，输出的文件就是排好序的。

流程第5步要遍历所有inode，复制对应的文件。复制过程分为两大步：

1. 复制文件数据
2. 建立新的间接索引

按照这个顺序执行，那么碎片整理后的硬盘结构（data region）如下：

`|0的数据块|0的间接索引块|1的数据块|1的间接索引块|...|n的数据块|n的间接索引块|空闲块|空闲块|...|`

上述两步存在一些共性，体现在：

- inode结构相同。
  - 例如：都要先处理10个直接索引块，再处理4个间接索引块，之后是1个2级间接索引和1个3级间接索引。
- 对间接索引的处理方式都有递归性

因此，这两步的函数接口是类似的，函数的定义也大同小异。

- 接口函数分别为 **copy_data()** 和 **write_inode()** 。接口函数会在 **main()** 中被调用。主要负责适配inode结构：对inode中每个直接索引或间接索引块，调用递归函数。
- 这两个接口调用的递归函数分别为 **copy_block()** 和 **write_iblock()** 。递归函数负责实际地写入输出文件，并递归地处理间接索引。

下面是递归函数的代码（以 **copy_block()** 为例）。

```
static int copy_block(int level, int idx, int tot) {
    if (tot <= 0)
        return 0;       // block unused
    if (level == 0)
        return copy_db(idx);     // direct block
    // read indirect block
    fseek(fin, idx2addr(idx), SEEK_SET);
    fread(iblock[level], super->size, 1, fin);
    int cnt = 0;
    for (int i = 0; i < ib_tot_item; ++i)
        cnt += copy_block(level - 1, iblock[level][i], tot - cnt);
    return cnt;
}
```

- 递归终止条件是到达直接索引块，此时直接复制这一块（通过 **copy_db()** 实现）。
- 对间接索引块，首先读出间接索引块的内容，对其中的每一个索引，递归调用。
- **level** 表示当前所在的间接索引层数。

下面是接口函数的代码（以 **copy_data()** 为例）。

```
static int copy_data(INODE *in) {
    // number of blocks contained in file *in (round up)
    int tot = (in->size + super->size - 1) >> block_shift
    int cnt = 0;
    for (int i = 0; i < N_DBLOCKS; ++i)
        cnt += copy_block(0, in->dblocks[i], tot - cnt);
    for (int i = 0; i < N_IBLOCKS; ++i)
        cnt += copy_block(1, in->iblocks[i], tot - cnt);
    cnt += copy_block(2, in->i2block, tot - cnt);
    cnt += copy_block(3, in->i3block, tot - cnt);
    return cnt;
}
```

- 该接口函数又可以视作是递归函数的入口函数。为Inode结构体中的各个成员调用递归函数，并正确的传参。
    - 例如，对于每个直接索引块，递归 **level** 为0，表示直接索引， **idx** 被定位到inode中的某个具体的值。
- 除了调用递归函数之外，该函数还负责控制
- 该函数结构清晰，直接对应到 **Inode** 结构体。因此，若要修改 **Inode** 为其他结构，这里只需要简单适配即可。
    - 另外一组函数也是类似的，只是传参略有不同。

函数实现中还有一些细节，例如每个参数的含义，返回值含义等。这些内容都在注释中详细(?)说明了，这里不再赘述。

结果自检

实现在 disk_dump.c 中。

实现了输出空闲块链表，文件内容，索引块内容等功能。从而可以方便地查阅硬盘内的结构。

输出文件或索引块时也是递归执行的，代码逻辑与碎片整理中的递归函数基本一致。

基于该程序可以实现一个简单的自检，实现在 test.sh 中：

- 调整 disk_dump.c 的代码，使其按顺序打印硬盘中每个文件的内容
- 分别对原磁盘和碎片整理后的磁盘调用 disk_dump.c
- 通过shell命令 `cmp` ，比较输出结果
    - 若结果完全一致（此时 `cmp` 无输出），那么至少能说明，碎片整理后，硬盘的文件结构依然保持完整：
        - 文件都能正常访问，直接和间接索引的建立是正确的。
        - 不存在文件丢失，每个文件都完整的复制到了新的硬盘中。

当然，碎片整理还有一些其他内容需要检查，例如空闲块链表组织结构等。这可以通过修改 disk_dump.c ，肉眼观察得到。

可能的改进

对硬盘进行碎片整理，通常是原地进行的？因此可能的改进是：把本程序修改为在原硬盘上直接进行碎片整理，而不是输出为一个新的硬盘。