CS355 Hw 6

Due by the end of day **Sunday, 4/7**. Supplementary questions due by **Monday 4/8 in class**.

Programming Assignment: Defragmentation

In this project, you will implement a disk defragmenter for a Unix-like (inode-based) file system. File system defragmenters improve performance by compacting all the blocks of a file into sequential order on disk.

Detailed specifications:

- **Overview**:
  - Your defragmenter will be invoked as follows:
    `./defrag <fragmented disk file>`
  - Your defragmenter should output a new disk image with "-defrag" concatenated to the end of the input file name. For instance,
    `./defrag myfile`
    should produce the output file "myfile-defrag".
  - All the above should be explained in a man page that can be pulled up with:
    `./defrag -h`
- **Data Structures**: There are two important data structures stored on disk: the *superblock* and the *inode*.
  - **superblock**

    ```
    struct superblock {
        int size; /* size of blocks in bytes */
        int inode_offset; /* offset of inode region in blocks */
        int data_offset; /* data region offset in blocks */
        int swap_offset; /* swap region offset in blocks */
        int free_inode; /* head of free inode list, index, if disk is full, -1 */
        int free_block; /* head of free block list, index, if disk is full, -1 */
    };
    ```

    On disk, the first 512 bytes contain the boot block (and you can ignore it). The second 512 bytes contain the superblock. All offsets in the superblock start at 1024 bytes into the disk and are given as blocks.

    For instance, if the inode offset is 1 and the block size is $512B$, then the inode region starts at $1024B + 1 \times 512B = 1536B$ into the disk.

    Each region fills up the disk to the next region; the swap region fills the disk to the end.
  - **inodes**

```
#define N_DBLOCKS 10
#define N_IBLOCKS 4
struct inode {
    int next_inode; /* index of next free inode */
    int protect; /* protection field */
    int nlink; /* number of links to this file */
    int size; /* numer of bytes in file */
    int uid; /* owner's user ID */
    int gid; /* owner's group ID */
    int ctime; /* change time */
    int mtime; /* modification time */
    int atime; /* access time */
    int dblocks[N_DBLOCKS]; /* pointers to data blocks */
    int iblocks[N_IBLOCKS]; /* pointers to indirect blocks */
    int i2block; /* pointer to doubly indirect block */
    int i3block; /* pointer to triply indirect block */
};
```

The inode region is effectively a large array of inodes. An unused inode has 0 in the `nlink` field and `next_inode` field contains the index of the next free inode. For inodes in use, the `next_inode` field is not used.

The size field of the inode is used to determine which data block pointers are valid. If the file is small enough to fit in `N_DBLOCKS` blocks, then the indirect blocks are not used. Note that there may be more than one indirect block. However, there is only one pointer to a doubly indirect block and one pointer to a triply indirect block. All block pointers are relative to the start of the data region. You may assume 4-byte integer size for block pointers in this file system.

The free block list is maintained as a linked list. The first 4 bytes of a free block contain an integer index to the next free block; the last free block contains -1 for the index.

- **Details** You will be given a disk image containing a file system. It will be correct (no corruption), to the specification described above.

  You should read in the disk, find inodes containing valid files, and write out a new image containing the same set of files, with the same inode numbers, but with all the blocks in a file layed out contiguously. Thus, if a file originally contained blocks $6, 2, 15, 22, 84, 7$ and it was relocated to the beginning of the data section, the new blocks would be $0, 1, 2, 3, 4, 5$.

  After defragmenting, your new disk image should contain:

  - the same boot block (just copy it)
  - a new superblock with the same list of free inodes but a new list of free blocks (sorted from lowest to highest to help prevent future fragmentation)
  - new inodes for the files

– data blocks at their new locations

A sample disk image for you to work with is available at `~dxu/handouts/cs355/datafile-frag`.

Reminder that you will need to do binary file I/O to read in the datafiles. You can do this with the `fread` library function.

- **Testing & Extra Credit** Effective testing for this assignment will require that you create a variety of fragmented disks according to the specification. Note that `datafile-frag` does not contain large enough files to use double indirect or triple indirect pointers, thus generating test disks with larger files that exercise double and triple indirect blocks is highly recommended. I am not requiring that everyone generate disks. If you do, extra credit will be granted based on the complexity of your test disks. You are also encouraged to post your defragmented disks to the course github repo. For this assignment the easiest way to be sure of your defragmenter's correctness is to compare your output disk to someone else's. Two identically defragged disks are a pretty good indication of correctness. There are other more involved ways, but they will require writing additional tracers/testers. Therefore, please help each other out! As usual, please name your disks accordingly and clearly explain its purpose/layout in a README.

Supplementary Questions: Do the following questions from Tanenbaum.

- pages 333-336, problem numbers 20, 21, 24, 28, 30, 40 and 41.

Deliverables:

1. Hardcopy: Answers to the supplementary questions.
2. Electronic submissions of the programming assignment.