Project 5: Defragmentation

Overview

This assignment has several goals. First, you'll get experience with file system structure. Second, you'll write a offline defragmentation algorithm. Third, you'll practice writing code without memory leaks. Finally, you'll practice describing an implemented algorithm with prose.

File system defragmenters can improve performance by laying out all the blocks of a file in sequential order on disk. In this project, you will write a disk defragmenter for a Unix-like file system.

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 datafile
```

should produce the output file "datafile-defrag".

Readings

It will be useful to read OSTEP Chapter 39, Chapter 40.

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 bytes 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 */
    int free_iblock; /* head of free block list */
};
```

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*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; /* list for free inodes */
    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; /* time field */
    int mtime; /* time field */
    int atime; /* time field */
    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 zero 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, this 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.

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.

Program Specifications

You will be given a disk image containing a file system. It will be correct (no corruption), and the free list of the superblock will list all the free indoes and the free data blocks.

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 here.

Hints

Binary File I/O

You will need to do binary file I/O to read in the datafiles. You can do this with the fread library function. Here's some sample code to get you started. Note that this code does no error checking; if you use this code, you'll need to update it.

```
FILE * f;
char * buffer;
size_t bytes;
buffer = malloc(1024);
f = fopen("datafile-defrag", "r");
bytes = fread(buffer, 1024, 1, f);
```

Hand In

When we grade your project, we will expect to see the following files in your repository (there may be more):

- Source code and header files that implement your defragmenter.
- A Makefile to build your code. Typing make should build defrag.
- defrag should be compiled with flags: `-Wall`
- A file called README which describes the algorithm you have used to implement defragmentation. Your README may optionally include any additional notes on your program that you think are important. The README is **NOT** optional for this project.

You should not turn in any binary or .o files. Only turn in source code and header files.

Grading

When your code compiles (using the flags listed above), gcc should not display any warnings. Any warnings that remain will negatively impact your grade.

We will grade your code primarily based on **how well the implementation works**. That is, we will run your code through a suite of inputs (fragmented disk images); your grade will largely be based on how many of these disk images you program correctly defragments. However, you are more likely to get partial credit if you have well-structured, commented code. If we cannot discern what your code was supposed to do, it will be difficult to assign you partial points if your program does not work as expected.

However, we will also be evaluating your memory usage. For this project, your program should have no memory leaks or memory that is still reachable when the program exits. To verify this, we will run your program with valgrind:

```
% valgrind --leak-check=full --show-reachable=yes defrag <fragmented disk file
```

You should use the same command to verify that you are correctly freeing all allocated memory. When you are successful, you should see the following message:

```
All heap blocks were freed -- no leaks are possible
```

Adapted from WISC CS537 by Emily Gember-Jacobson