OSLAB Project V Report

Yao Shunyu 10152130134
Lv Yingzhe 10152130255

December 30, 2017

1 Introduction

In this project, we are required to design a file system defragmenter, which improves file system performance by laying out all the blocks of a file in sequential order on disk. Our defragmenter is for a quiet ancient unix-liked file system, called AFS (Andrew File System). By completing this project, we got a sense of file system structure and memory-leak-free programming technique.

2 Usage

2.1 Defragmenter

Our submit version only contains defragmenter source files, you should build execution first, just type the following commands in your terminal:

```
% cd <src folder>
% make
```

After that, you will get a executable file called `defrag`, you can call it like this:

```
% ./defrag <fragmented disk file>
```

Our output file will be named with `-defrag` suffix concatenated after the input file name but before its extension name (if any).

2.2 Validation

If you want to verify the correctness of our output result, we also provide validation program. What you only need to do is modify the **50th line** in `main.c`:

delete `//` before `validation(inFile);`

After that, save the source file, re-build and execute new `defrag` program, this time **with the file name need to be verified as argument**. After that, you'll get all files in this file image in `./unpacked` folder, and get debug infos from your terminal.

3 Algorithm

When considering how to reformat data blocks, we figured out two way to complete it:

- **File Sequential Method**:
  For a single file, place its data and indirect blocks continuously and in the order of sequentially reading. All files are placed one by one from the beginning of data region.
- **Two Region Method**:
  Place all data blocks of files (in sequentially reading order and continuously) one by one from the beginning of data region, namely **data block region**. Then put indirect blocks of all files (also in sequentially reading order and continuously) one by one after data block region, namely **indirect block region**.

As a result, we choose to use **Two Region Method**, the analysis will be shown in the following two subsections.

3.1 File Sequential Method

Under this method, both **sequential and small file random access speed** will be improved, but **random access speed for big files** is bad. For sequential reading, since the next block need to read always under the magnetic head, minimal movement is required, which means the sequential access time is minimized. For random access in small files, this method also performs well. If the block need to access is indexed by a direct block, magnetic head only need to skip over a few data blocks, no seeking time in most cases. If the block is indexed by a first indirect block, magnetic skip over a few data block to access I1 block, then skip over at most $\frac{512}{4} = 128$ blocks to access the data block. However, for blocks indexed by a second indirect block, such number is $\frac{512}{4}^2 = 16384$, for third indirect indexed block, $\frac{512}{4}^3 = 2097152$, not to mention the time to find I2 or I3 block (other indirect blocks needed are accessed half-way), that's a huge cost!

3.2 Two Region Method

This method is just reversed, **big file random access speed** will be improved, but **sequential and small file random access speed** is slow. Considering the same example in the previous subsection, for accessing a third indirect indexed block, after read I3 block, our pity magnetic head need only to across a few indirect blocks to access I3 block, then I2 block, then I1 block, then wired back to data block region for data block. However, for direct block or first indirect indexed block, and for sequential reading, magnetic head also need to wire back and forth unnecessarily, which makes it slower that our first method.

3.3 Overall Algorithm

Considering the great portion takes up by small files in most cases, we think reduce small file access speed is more important than big ones. Therefore, we choose **file sequential method** as our defragment algorithm. After choosing reformat method, we can describe our overall algorithm now. We execute the following procedures respectively:

1. Copy boot block into output file
2. Read super block from input file and calculate necessary values
3. Hold place for super block and i-nodes (they need to be modified so cannot write right away)
4. Read i-node and data blocks of a file one by one, then reformat and write into output file, also modify i-node and write back

5. Read data free list, sort its index, then write back free data blocks (we copy free block content from input file for the convenience of test, although not necessary)

6. Copy swap region into output file

We guarantee that our output file is correct if the input file has correct file structure.

4 About the Sample File

We found that the sample file system given in specification has some errors, decided to list them down and give our remedies.

4.1 Missed Blocks in Data Region

In the sample file, the swap region offset is 10243 and data region offset is 4, so the data region should contains 10239 blocks. However, data blocks indexed between 10135 to 10238 are not data blocks used by files nor free block recorded in free-list. To make output file completion, we simply copy these blocks into output file (we don't add them into free list since it may affect validation by TA).

4.2 Total Block Number

After make up lost blocks, we found another problem, the file size is not correct. Since the swap region offset is 10243, this file must contains $2 + 10243 = 10245$ blocks, say $10245 * 512B = 5245440B$, but input file only has $5243392B$, in other word, 4 blocks are mysteriously lost. The only thing we can do is modify our previous remedy, let it fill file blocks into output file until data region ends (even origin file is shorter, our remedy can copy obsolete blocks into output file instead).