CSC501 Fall 2019

PA3.B: A Defragmenter

Due: December 7 2019, 4:00 AM

The old Unix file system can get decreasing performance because of fragmentation. File system defragmenters improve the performance by laying out all blocks of a file in a sequential order on disk, and we are going to implement one in this homework.

Disk structure

You will be given a disk image containing a file system, and it will be correct, i.e., no corruption. Two major data structures that are used to manage this disk are related to the defragmenter: the superblock and the inode. The superblock structure is given below:

```
struct superblock {
  int blocksize; /* 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 */
  int free_block; /* head of free block list */
}
```

On disk, the first 512 bytes contains the boot block, and its actual format is not relevant to us. The second 512 bytes contains the superblock, the structure of which is defined above. All offsets in the superblock are given as blocks. Thus, if the inode_offset is 1 and the blocksize is 1 KB, the inode region starts at address 1024B + 1*1KB = 2KB into the disk. Each region fills up the disk up to the next region, and the swap region fills the disk to the end.

The inode region is effectively a large array of inodes, and the inode structure is given below:

```
#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;  /* Number 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 */
 };
```

An unused inode has zero in the nlink field and the next_inode field contains the index of the next free inode, and the last free inode contains -1 for the index. For inodes in use, the next_inode field is not used. The head of the free inode list is a index into the inode region. The inodes in the inode region are all contiguous. Independent of block boundaries, the whole region is viewed as one big array. So it is possible for there to be an inode that overlaps two blocks.

The size field of the inode is used to determine which data block pointers are valid. If the file is small enough to be fit in N_DBLOCKS blocks, the indirect blocks are not used. Note that there may be more than one indirect block. However, there is only one pointer to a double indirect block and one pointer to a triple indirect block. All block pointers are indexes relative to the start of the data block region.

The free block list is maintained as a linked list. The first four bytes of a free block contain an integer index to the next free block; the last free block contains -1 for the index. The head of the free block list is also a index into the data block region.

Your task: the defragmenter

You should read in the disk image, 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 laid out contiguously. Thus, if a file originally contains blocks {6,2,15,22,84,7} and it is relocated to the beginning of the data section, then the new blocks would be {0,1,2,3,4,5}. If there are indirect pointer blocks, the block containing pointers need to be before the blocks being pointed to, and this applied to all level of indirect points.

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 prevent future fragmentation), new inodes for valid files, and data blocks at their new locations.

You do not need to copy/save the contents of free blocks when you create the defragmented disk. They can be left as they are, initialized to zeros, or anything else as long as the free list itself is well constructed, i.e., it is a valid linked list.

A sample disk image disk_frag_0 is provided for you to work with. The output disk image should be named as "disk_defrag". Your program should take one parameter, i.e., the name of the disk image. You will need to do binary file I/O to read in the datafiles. You can do this with the fread() C library function. Here is some sample code:

```
FILE * f;
unsigned char * buffer;
size_t bytes;
buffer = malloc(10*1024*1024);
f = fopen("disk_frag","r");
bytes =  fread(buffer,10*1024*1024,1,f);
```

Please do not use this directly, as it does no error checking. If you want to find out how big a file is from within your program, you can use the fstat() function. You can change the type of the buffer if you see a fit.

To help you understand how to do cast in C and read the disk image, we provide the following code to show how to read the superblock:

```
superblock *pSB = (superblock*) &(buffer[512]);
... pSB->blocksize ...;
... pSB->inode_offset ...;
```

You can also use the following code:

```
superblock sb;
sb.blocksize = *((int *) &(buffer[512]));
sb.inode_offset = *((int *) &(buffer[512 + 4]));
```

Try to make sense of the code above, from which you should get an idea on how to read the disk image. Note the code above assuming the buffer is of the "char *" or "unsugned char *" type. If you choose to declare your buffer to something else, e.g., "int *", you need to adjust the code above.

To avoid potential compiler-induced problems in the code above, please add "-O0" into your compiler flags. Also please check (1) the address buffer returned by malloc can be divided by four and (2) sizeof(struct superblock) is 24 and sizeof(struct inode) is 100. If a violation is found, please let the instructor know.

A different and safer approach of reading a word from an arbitrary address is as following:

```
superblock sb;
int readIntAt(unsigned char *p)
{
   return *(p+3) * 256 * 256 * 256 + *(p+2) * 256 * 256 + *(p+1) * 256 + *p;
}
sb.blocksize = readIntAt(buffer + 512);
sb.inode_offset = readIntAt(buffer + 512 + 4);
```

Disk images: disk_pairs.zip.

The input disk_frag_1 is similar to disk_frag_0 with the following constraints, which are followed by disk_frag_2 and disk_frag_3:

1. All unused inodes have -1 for the last four index-related pointer fields and 0 for other fields after next_inode. For inodes in use, index-related pointer fields that are not used should have values -1.
2. A data block used as a indirect pointer block will have -1 for all the unused pointers beyond file size.
3. A free data block will contain only 0 after the first four bytes for pointing to the next free data block. A data block in use will contain only 0 beyond the file size.
4. Unused bytes in the super block and the inode region are 0. For the inode region, if the total size is 1024 bytes, then the last 24 bytes are unused bytes, as each inode is 100 bytes and the inode region can hold 10 inodes

If you want to get the exact output images as provided, the rules above should be followed. Since the input images already follows the rules, you do not need to change your code much to make the output images satisfy the rules.

After you finish, turn in your source code, README, and Makefile. You can name your source files any names you want, but the executable generated by the Makefile should be "defrag".

Bonus points (up to 5 points):

If you have a disk image that can catch errors the provided three cannot catch, you can submit the input image (limit to 1 image) and the expected output image. This means that there exists an implementation that is able to generate the expected disk images for the three provided ones, but you have a valid disk image that can make the implementation crash or generate a wrong disk image. If your implementation satisfies this, then it directly shows. Otherwise, you need to argue using words. Your image pair has to follow the rules mentioned above.

If you choose to submit such images, you also need to (1) describe what problems you can catch that are missed and (2) submit your images and any program you may write to create your image. A valid pair can get up to 20 bonus points for this homework.

Q & A:

1. Q: How to determine whether an inode is current being used?
   A: If its nlink has 0; you can also check whether it is in the free list, but this will be more complicated than checking nlink; since inodes not in use also have the size field as 0, you can not use it to determine whether an inode is used.
2. Q: Should I do the defragmentation within the original data block region?
   A: You can, but you do not have to. A simpler way is you read data blocks from the old disk image and then write the data blocks, including those storing pointers, onto the new disk image.
3. Q: In which order should we traverse the inodes?
   A: Treat the whole inode region as an array, start from index 0, and skip the ones that are free.

Back to the CSC501 web page