Operating Systems Practical Assignment – Filesystem Compaction

Erik van Zijst and Sander van Loo
August 19th, 2004

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{erik,sander}@marketxs.com

Introduction

This document describes our work for the second practical assignment of the operating systems course. For this assignment we had to implement a user program that can defragment minix filesystems. Since files are stored in fixed-sized chunks, called blocks, fragmentation can occur when the individual blocks of a file are not adjacent to each other on disk. The effect of filesystem fragmentation is that the disk's heads must often be moved in order to read all the file blocks, making disk access slower than when all blocks were adjacent.
In this text we will describe the features and the development process of our defragmentation program. We will also describe how we tested the program and how the results matched our expectations. We will describe the development phase, the testing phase and our conclusions in separate chapters.

Program Development

Since the defragmentation is entirely implemented as a normal user program, no modifications to the minix kernel were made.
The program we implemented is called `defrag` and takes two filesystem arguments. The first argument is used as the input filesystem, the second as the output filesystem. Both must be minix V2 filesystems. Other types (including V2 with reversed byte ordering) are not supported. The program reads the input filesystem (hereafter referred to as *oldfs*) and copies everything to the output filesystem (hereafter referred to as *newfs*) in a structured way. It visits each file on the fragmented filesystem and copies all of its blocks to *newfs* in adjacent order. The exact order of a file's data blocks is as described in the assignment. The order in which the individual files and directories are written to disk is arbitrary since files are processed in the order of their inode numbers. Our program expects the destination to be a newly created, empty minix filesystem that has enough space to store all files on the fragmented filesystem.
When the program starts it opens both filesystems, checks their type and checks whether *newfs* is empty. It then counts the number of used data blocks on *oldfs* and check whether *newfs* has at least that many data blocks. This means that in principle it is possible to write the defragmented filesystem that is only partially filled, to a disk that is smaller than the original, but large enough to store all files. The program also runs a check on the number of inodes. Since the program does not change the location of the inodes, it checks whether *newfs* has at least as many inode slots to copy all used inodes of *oldfs*. The consequence of this approach is that the address of the last inode on *oldfs* must also be available on *newfs* (as the inode space is not compacted). In practice this means that partially filled large filesystems can only be defragmented to smaller filesystems if their inodes do not occupy inode blocks that are not available on the smaller *newfs*.
Before defragmentation commences, the program verifies that both filesystems are unmounted. Reading and writing mounted filesystems that are in use can easily lead to corruption or inconsistent reads. If one of the filesystems is mounted, the program aborts. These checks are implemented in the functions `fs_open()`, `fs_check_mtab()` and `fs_is_empty()`. Although *newfs* is expected to be a newly created, consistent filesystem, the program does detect certain inconsistencies in *oldfs* during defragmentation. Detected inconsistencies include pointers in inodes that refer to unallocated blocks in the zone map.

The actual defragmentation algorithm, implemented in the functions `fs_defrag()`, `fs_defrag_dblock()`, `fs_defrag_idblock()` and `fs_defrag_didblock()`, works its way through the fragmented filesystem in iterative loops. Recursion is deliberately avoided. Although a clean approach to filesystem traversal may be to read the contents of a directory (starting at the root directory), process all files and then recursively process all subdirectories, it may fail on filesystems that have a deeply nested directory structure. In some of these cases, the directory structure may require more stack space than the program has available. Also, when a lot of variables are allocated on the stack, this can significantly reduce the maximum stack depth during program execution. To avoid recursion problems, we implemented the filesystem traversal algorithm as an iterative loop. The main loop can be found in `fs_defrag()`. Inside the loop, the algorithm examines the inode zones of *oldfs* and processes each inode entry that is in use (i.e. has its bit set in the inode map). When the inode is read, the algorithm examines all of its 7 direct zones. For each zone pointer that is set (i.e. points to a block, rather than `NO_BLOCK`), the method `fs_defrag_dblock()` is called. This method copies the data block from *oldfs* to *newfs*. The location on *newfs* is determined by the value of `newfs_next_free`. This block pointer is used throughout the defragmentation process and is incremented every time a data block is written to *newfs*. The new block location is then stored in the inode. After the direct zone pointers, the indirect zone is examined. If it is in use (not equal to `NO_BLOCK`) , it is passed to `fs_defrag_idblock()`. This function reads the block pointers from the single indirect block and each used pointer is passed to `fs_defrag_dblock()`. Similarly, the double indirect block pointer in the inode is processed by `fs_defrag_didblock()` which calls `fs_defrag_idblock()` for each used block pointer in the indirect data block.
After all zone pointers have been processed, the inode that was stored and altered in memory during the process, is written to the inode map on *newfs* in the same slot as it occupied on *oldfs*.
Inside the main defragmentation loop is a check for inode-type. Since only regular files and directories have a dynamic size, only these two types use the inode zone pointers to point to data blocks. Other file types such as character devices store other information in the zone pointers. It is important to distinguish between these types to avoid examining zone pointers that are not used to point to blocks.
After all inodes are processed, the program stores the inode and zone bitmap on *newfs* as they were manipulated in-memory during the defragmentation process.

Experiments

To verify the correct operation of our defragmentation program, we ran a number of experiments to test it under various circumstances.

For our first experiment we created two equally sized 1.44MB filesystems. We copied several files to the first filesystem, scattered over several directories. We then unmounted it and used the disk editor to check which files had its data blocks at the start of the disk. We removed them and copied larger files to the disk. This caused the first blocks of these new files to be stored in the beginning of the disk in the blocks that were freed by removing the previous files, while the rest of the blocks are stored behind the blocks of the older files. To make things worse, we removed another old file, leaving some unallocated blocks at the beginning of the disk. We verified this using the disk editor and concluded that our filesystem was indeed fragmented. The unallocated block that was created can easily be seen by inspecting the filesystem's zone bitmap. The first part of the zone bitmap is depicted below. The first bitchunk shows the unallocated block.

```
Device = /dev/hd1a       V2 file system
Block  =     3 of 1440   Zone bit map
Offset =     0           Block 33 of 1440

>     0    65527            w?
      2    65535            ??
      4    65535            ??
      6    65535            ??
      8    65535            ??
     10    65535            ??
     12    65535            ??
     14    65535            ??
     16    65535            ??
     18    65535            ??
     20    65535            ??
     22    65535            ??
     24    65535            ??
     26    65535            ??
     28    65535            ??
     30    65535            ??
```

After defragmentation of `/dev/hd1a` to `/dev/hd1b`, the disk editor reports no holes in the zone bitmap of the new filesystem as depicted below.

```
  Device = /dev/hd1b       V2 file system
  Block  =     3 of 1440   Zone bit map
  Offset =     0           Block 33 of 1440

  >     0    65535           ??
        2    65535           ??
        4    65535           ??
        6    65535           ??
        8    65535           ??
       10    65535           ??
       12    65535           ??
       14    65535           ??
       16    65535           ??
       18    65535           ??
       20    65535           ??
       22    65535           ??
       24    65535           ??
       26    65535           ??
       28    65535           ??
       30    65535           ??
```

In addition to the compacted zone map, we checked the consistency of the new filesystem and compared its output to the original filesystem:

```
# fsck -a /dev/hd1a

Checking zone map
Checking inode map
Checking inode list

blocksize =  1024       zonesize  =  1024

     5    Regular files
     2    Directories
     0    Block special files
     1    Character special file
   472    Free inodes
     0    Named pipes
     0    Symbolic links
   721    Free zones
# fsck -a /dev/hd1b

Checking zone map
Checking inode map
Checking inode list

blocksize =  1024       zonesize  =  1024

     5    Regular files
     2    Directories
     0    Block special files
     1    Character special file
   472    Free inodes
     0    Named pipes
     0    Symbolic links
   721    Free zones
#
```

Clearly, both filesystems contain the same number of files and directories. Also note the character device we created. We repeated the test with a destination filesystem that was much larger (70MB versus 1.44MB). Although `fsck` shows that the new filesystem has much more free zones and inodes, it does contain a correctly defragmented version of the original filesystem. The output of `fsck` is given below:

```
# fsck -a /dev/hd1c

Checking zone map
Checking inode map
Checking inode list

blocksize =  1024       zonesize  =  1024

     5    Regular files
     2    Directories
     0    Block special files
     1    Character special file
 11671    Free inodes
     0    Named pipes
     0    Symbolic links
 68571    Free zones
#
```

To verify whether our program writes a file's data blocks to *newfs* in the correct order, we added some debugging output to `defrag` that prints the block type of every data block it writes to the disk. When we let `defrag` write a large file that used both its single and double indirect block, the following output appears:

```
data block (35)
data block (36)
data block (37)
data block (38)
data block (39)
data block (40)
data block (41)
indirect block (42)
data block (43)
data block (44)
...
data block (297)
data block (298)
double indirect block (299)
indirect block (300)
data block (301)
data block (302)
...
data block (555)
data block (556)
indirect block (557)
data block (558)
data block (559)
...
data block (607)
data block (608)
```

Note that we have removed some lines for readability, but it clearly shows how `defrag` first writes the file's 7 direct data blocks, then the single indirect block that contains 256 pointers, followed by the actual data blocks pointed to. After the data blocks of the single indirect block comes the double indirect block, followed by the first single indirect block that the double indirect block points to. Following are the 256 data blocks of the single indirect block. Next comes the second single indirect block that is pointed to by the double indirect block. It is followed by its last 51 data blocks. We left this debugging feature in the program. It can be switched on by recompiling the program after adding `-DDEBUG` to the compilation flags in the `Makefile`.

To test if our defragger handles files with holes properly, we created a file with a hole on our test filesystem by copying the contents of the file 'profile' to a new file named 'hole' with `dd`. We instructed `dd` to start writing to the new file after skipping 1 block (1024 bytes) in the new file:

```
# dd if=profile of=hole bs=1k seek=1
```

The disk editor (`de`) shows that the first direct zone (zone 0) is indeed `NIL` and that actual data is stored only in the block referenced by zone 1:

```
Device = /dev/hd1a       V2 file system
Block  =     4 of 1440   I-nodes
Offset =   448           I-node 8 of 480  (in use)

>   448    33188            regular  ---rw-r--r--
    450        1            links 1
    452        0            user root
    454        0            group operator
    456     1177            file size 1177
    458        0
    460    27665            a_time Fri Jul 30 17:41:05 2004
    462    16650
    464    27665            m_time Fri Jul 30 17:41:05 2004
    466    16650
    468    27665            c_time Fri Jul 30 17:41:05 2004
    470    16650
    472        0            zone 0
    474        0
    476       36            zone 1
    478        0
```

Looking at the filesystem after defragmentation shows us that the hole is preserved; zone 0 is still `NIL` and zone 1 now references a different data block (caused by the defragmentation process):

```
Device = /dev/hd1b       V2 file system
Block  =     4 of 1440   I-nodes
Offset =   448           I-node 8 of 480  (in use)

>   448    33188            regular  ---rw-r--r--
    450        1            links 1
    452        0            user root
    454        0            group operator
    456     1177            file size 1177
    458        0
    460    27665            a_time Fri Jul 30 17:41:05 2004
    462    16650
    464    27665            m_time Fri Jul 30 17:41:05 2004
    466    16650
    468    27665            c_time Fri Jul 30 17:41:05 2004
    470    16650
    472        0            zone 0
    474        0
    476      601            zone 1
    478        0
```

To explicitly test if our defragmentation process leaves other inode types (such as character devices) in tact we created a character device with the `mknod` command:

```
# mknod chardev c 4 16
```

Looking at the inode with the disk editor (`de`) shows that the zone 0 pointer contains data:

```
Device = /dev/hd1a       V2 file system
Block  =     4 of 1440   I-nodes
Offset =   128           I-node 3 of 480  (in use)

>   128     8612            character  ---rw-r--r--
    130        1            links 1
    132        0            user root
    134        0            group operator
    136        0            file size 0
    138        0
    140    26261            a_time Fri Jul 30 17:17:41 2004
    142    16650            major 4, minor 16
    144    26261            m_time Fri Jul 30 17:17:41 2004
    146    16650
    148    26261            c_time Fri Jul 30 17:17:41 2004
    150    16650
    152     1040            zone 0
    154        0
    156        0            zone 1
    158        0
```

After defragmentation the contents of the inode are preserved as is shown with the disk editor:

```
Device = /dev/hd1b       V2 file system
Block  =     4 of 1440   I-nodes
Offset =   128           I-node 3 of 480  (in use)

>   128     8612            character  ---rw-r--r--
    130        1            links 1
    132        0            user root
    134        0            group operator
    136        0            file size 0
    138        0
    140    26261            a_time Fri Jul 30 17:17:41 2004
    142    16650            major 4, minor 16
    144    26261            m_time Fri Jul 30 17:17:41 2004
    146    16650
    148    26261            c_time Fri Jul 30 17:17:41 2004
    150    16650
    152     1040            zone 0
    154        0
    156        0            zone 1
    158        0
```

To be sure that not only the filesystem's structure is maintained after defragmentation, but that the contents of the actual files are untouched, we created a script that we ran after every defragmentation. This script traverses the original filesystem, computes the checksum of every file and directory (using `cksum`) and compares it with the checksum of the files on the new filesystem. The script was unable to find differences after our tests. The script can be found at `/usr/src/extra/compare.sh`.

Conclusions

In this text we describe our efforts to build a defragmentation utility that is implemented as a normal user-space program and can defragment an unmounted filesystem to an empty filesystem that has sufficient space. We described the program's internal structure and exposed it to a number of tests. After every test we inspected the new filesystem using tools including `fsck` and `de`. We also mounted the filesystems to compare the checksum of all files with the original filesystem. In all cases the defragmentations were successful.