Operating Systems practical 2008/2009 documentation for assignment 2 – file system compaction

Richard van Heuven van Staereling
richard@few.vu.nl

Marc Hage
mrhage@few.vu.nl

August 24, 2009

Development

One design problem involves the numerous types. We are working with bytes, block numbers, zone numbers; which type should we use for offsets? Also, for a data block, is the offset from the beginning of disk or from the first data sector? It makes sense to use block numbers wherever possible and talk bytes only on the lowest level (when reading/writing from disk). Ultimately, we have decided to create a couple of `_OFFSET`, `_SZ` and `_BLOCKS` macros to make the code look neater: offset and size are in bytes, and everything is from beginning of disk.

Before this, we used the superblock info directly but the code became unreadable (e.g. `fs->superblock.s_imap_blocks`, now `IMAP_BLOCKS`). We considered copying the numbers to variables like `size_t block_sz` in the file system control block structure but this idea was short-lived because having the same information in multiple places is never a good idea. The only con to these macros is that most require the existence of an `fs`-variable.

While the defragmentation program does what it is supposed to do (compact the file system by removing fragmentation and storing files sequentially on disk), there is no control over the order of files. Since all files are to keep their inode-number, the obvious way to store files is in the order of their inode-number. The goal of defragmentation is performance improvement, but for a higher level of efficiency one could think, for example, of grouping files that are in the same directory also together on disk. Or, the access time of files could be taken into account when determining where files are stored.

Our implementation follows the standard "sorted-by-inode-number" approach but the above was taken into account to make it easy to extend the program with additional functionality. The order in which files are placed on `newfs` is decided by one for-loop (the one in `defrag_fs()`). By changing this order (for example, one could add inode numbers to an array while recursively traversing

the directories of the file system, and have the for-loop go over this array instead), it is possible to sort the files on disk in another way.

For checking the size of the output partition, we didn't simply check if their sizes are equal. The program should support the compaction of the old file system to a smaller file system; as long as there's enough space (so if `oldfs` is completely full, then yes, `newfs` needs to be at least as big). To check whether it is possible, our program calculates the space used in `oldfs` by counting the number of bits used in the zone bitmap. Looking at the amount of bytes used by all files is incorrect: we reasoned that the padding used at the end of zones needs to be taken into account as well since we can't use one zone for more than one file. The inode bitmap size is looked at as well. Since this is not compacted (all inodes keep their number), we only need to look at the last set bit in the inode bitmap. The new file system needs to have at least as many bits available.

Even though the assignment does not explicitly require the program to check whether the file systems are mounted or not, we did implement a check to make sure both are unmounted. While it should technically be possible to defragment a mounted file system, this would require extra security measures like file locking, or even better, catching requests to disk I/O and delaying them until it is safe. Since this is out of the scope of the assignment and the latter probably requires kernel modifications, we opted for a simple check: see if the filename of the file system is an entry in `/etc/mtab`.

From `mkfs.c`, a program that is required that the file system given as its argument is unmounted, we took inspiration for a `check_isunmounted()` function. This partially uses library functions from `<minix/minlib.h>`. However, if the given filename does not exactly match with the name used in the mount table, the program may continue even if it should abort. This can be demonstrated by using `mkfs` or `umount` in combination with a path containing `/./` somewhere; behavior afterwards is unpredictable. Since this is a trivial problem, no further attention is given to this, but it persists in our program. Corrections should be given in the form of improved library functions in the MINIX source code.

Verifying correctness

Comparing file contents

The most obvious way to test whether the program doesn't break the file system is to see if the file system structure is the same and that the contents are untouched. Rather than using commands like `ls` and `cat` to compare files (unfortunately there is no `fc` command in MINIX), we wrote a small program comparefs that recursively compares directories and their contents. Along with a call to `fsck`, this was added to the target `test` in the `Makefile` (so type `make test` to run both defrag and this test).

Handling holes in files

To test whether the program handles holes correctly, we created a file with a hole using the `dd`-command. The command we used was:

```
# echo "foobar" > nohole
# dd if=nohole of=hole bs=1024 seek=1
```

This created a file `hole` of size 1031 (1024 '\0'-characters + "foobar\n") with an inode that looks like this:

```
  Device = /dev/c0d2       V3 file system
  Block  =     4 of 1008   I-nodes
  Offset =   320           I-node 6 of 512  (in use)

  >   320    33188           regular  ---rw-r--r--                (base 10)
      322        1           links 1
      324        0           user root
      326        0           group operator
      328     1031           file size 1031
      330        0
      332    12681           a_time Wed Aug 12 03:05:45 2009
      334    19074
      336    12681           m_time Wed Aug 12 03:05:45 2009
      338    19074
      340    12681           c_time Wed Aug 12 03:05:45 2009
      342    19074
      344        0           zone 0
      346        0
      348       62           zone 1
      350        0
```

Zone 0 points to `NO_ZONE`. This should be preserved in the new file system and a check with `de` confirms this. We observed that the pointer to 0 is maintained and that zone 1 contains a different value (but this is expected).

Handling special files

To test whether the program handles unusual files correctly, we created a character device using the `mknod` command. The program should leave the inode intact and not allocate anything in the data space. The command for creating a character device:

```
# mknod blockdev b 12 34
```

The result can be seen by using `de` on this file system:

```
Device = /dev/c0d2       V3 file system
Block  =     4 of 1008   I-nodes
Offset =   256           I-node 5 of 512  (in use)

>   256    24996           block  ---rw-r--r--                    (base 10)
    258        1           links 1
    260        0           user root
    262        0           group operator
    264        0           file size 0
    266        0
    268     8629           a_time Wed Aug 12 01:58:13 2009
    270    19074           major 12, minor 34
    272     8629           m_time Wed Aug 12 01:58:13 2009
    274    19074
    276     8629           c_time Wed Aug 12 01:58:13 2009
    278    19074
    280     3106           zone 0
    282        0
    284        0           zone 1
    286        0
```

As you can see from the output above, an inode is created on the original file system. Observe that `de` recognizes this as a block device. Also note that the zone 0 space is 3106 (using output base hex, this is 0x0c22, with 0x0c=12 and 0x22=34, revealing that the zone 0 space is used to store the major and minor device).

To test our program, this inode needs to be left intact, and the zone 0 space should not be treated as a pointer to data. We used `de` to confirm that the copied inode is the same. We also put random data in the block supposedly pointed to in the zone 0 space and verified that this block is empty in the new file system. (Actually, block 3106 is out of range in this example and the program would have exited with a generic error, so we tested it after changing it to a usable value.) We also repeated the test with a character device.

Boundary checks

To make sure our program works as expected in all situations, we manually did some boundary checks. We did not write programs for this but edited the source partition manually using `de`, so therefore we will describe some of the tests we did here:

- changed the last bit in the inode map to 1 to check if all inodes are traversed and to check if non-existent inodes are recognized
- changed the first bit in the inode map's padding to 1 to check if it is ignored

- completely filled `oldfs`, decreased `s_ninodes` in superblock by 1 to test our size check
- completely filled `oldfs`, decreased `s_nzones` in superblock by 1 to test our size check

Handling other block sizes

As a bonus, we tried to implement the program in such a way that it is possible to handle other block sizes, on top of just the 1024-byte default. By not relying on constants but attempting to use the superblock variables as much as possible we have managed support bigger block sizes as well. If you compile the source normally it will enforce a size of 1024, but using `cc` with option `-DBLOCKSIZE=2048` (edit the `Makefile`) will get you a program enforcing a size of 2048, and so forth. The reason we added this as a compile option is to avoid failing any automated tests checking for this and because we would otherwise have to use `malloc()`. We tried to avoid the latter at all costs (but used the block size stored in the superblock as much as possible, of course asserting somewhere that they are equal).

The `de` is unable to handle block sizes other than 1024 but it is still possible to see the superblock and check out the contents in 1024-byte blocks (you just need to ignore `de`'s interpretation of the contents). We have discovered a couple of small bugs, although the program would have behaved correctly for a block size of 1024 bytes. It is still assumed the block size equals the zone size. Supporting that is not impossible but requires much more work (and increases risk of breaking things that work).