Righteous IT *Join the crusade!*

XFS (Part 3) – Short Form Directories

Posted on May 25, 2018May 31, 2018 by Hal Pomeranz

XFS uses several different directory structures depending on the size of the directory. For testing purposes, I created three directories– one with 5 files, one with 50, and one with 5000 file entries. Small directories have their data stored in the inode. In this installment we'll examine the inode of the directory that contains only five files.

```
0x000  49 4E 41 ED 03 01 00 00 00 00 00 00 00 00 00 00   INAí............
0x010  00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x020  5A EF 84 6C 11 16 6A BB 5A EF 84 67 03 38 E8 90   Zï.l..j»Zï.g.8è.
0x030  5A EF 84 67 03 38 E8 90 00 00 00 00 00 00 00 66   Zï.g.8è........f
0x040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x050  00 00 23 01 00 00 00 00 00 00 00 00 5A 52 79 A1   ..#.........ZRy¡
0x060  FF FF FF FF 27 FA 73 4A 00 00 00 00 00 00 00 01   ÿÿÿÿ'úsJ........
0x070  00 00 00 20 00 00 7F FC 00 00 00 00 00 00 00 00   ... ...ü........
0x080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x090  5A EF 84 47 13 FF 46 D3 00 00 00 00 04 08 E5 96   Zï.G.ÿFÓ......å.
0x0A0  E5 6C 3B 41 CA 03 4B 41 B1 5C DD 60 9C B7 DA 71   ål;AÊ.KA±\Ý`.·Úq
0x0B0  05 00 04 15 9F A1 0C 00 60 30 31 5F 73 6D 61 6C   .....¡ .`01_smal
0x0C0  6C 66 69 6C 65 01 04 17 97 9D 0A 00 78 30 32 5F   lfile..... .x02_
0x0D0  62 69 67 66 69 6C 65 01 04 17 97 9E 0C 00 90 30   bigfile..... ..0
0x0E0  33 5F 73 6D 61 6C 6C 66 69 6C 65 01 04 17 97 9F   3_smallfile.....
0x0F0  0A 00 A8 30 34 5F 62 69 67 66 69 6C 65 01 04 17    .¨04_bigfile...
0x100  A1 54 0C 00 C0 30 35 5F 73 6D 61 6C 6C 66 69 6C   ¡T .À05_smallfil
0x110  65 01 04 17 A1 55 00 00 00 00 00 00 00 00 00 00   e...¡U..........
0x120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x1A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x1B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x1C0  00 00 00 00 00 00 00 00 00 34 01 00 07 26 04 73   .........4...&.s
0x1D0  65 6C 69 6E 75 78 75 6E 63 6F 6E 66 69 6E 65 64   elinuxunconfined
0x1E0  5F 75 3A 6F 62 6A 65 63 74 5F 72 3A 61 64 6D 69   _u:object_r:admi
0x1F0  6E 5F 68 6F 6D 65 5F 74 3A 73 30 00 00 73 30 00   n_home_t:s0..s0.
```

We documented the "inode core" layout and the format of the extended attributes in Part 2 (https://righteousit.wordpress.com/2018/05/23/xfs-part-2-inodes/)of this series. In this inode the file type (upper nibble of byte 2) is 4, which means it's a directory. The data fork type (byte 5) is 1, meaning resident data.

Resident directory data is stored as a “short form” directory structure starting at byte offset 176, right after the inode core. First we have a brief header:

```
176      Number of directory entries                5
177      Number of dir entries needing 64-bit inodes   0
178-181  Inode of parent                            0x04159fa1
```

First we have a byte tracking the number of directory entries to follow the header. The next byte tracks how many directory entries require 64 bits for inode data. As we saw in Part 1 (https://righteousit.wordpress.com/2018/05/21/xfs-part-1-superblock/) of this series, XFS uses variable length addreses for blocks and inodes. In our file system, we need less than 32 bits to store these addresses, so there are no directory entries requiring 64-bit inodes. This means the directory data will use 32 bits to store inodes in order to save space.

This has an immediate impact because the next entry in the header is the inode of the parent directory. Since byte 177 is zero, this field will be 32 bits. If byte 177 was non-zero, then all inode entries in the header and directory entries would be 64-bit.

The parent inode field in the header is the equivalent of the usual “..” link in the directory. The current directory inode (the “.” link) is found in the inode core in bytes 152-159. The short form directory simply uses these values and does not have explicit “.” and “..” entries.

After the header come a series of variable length directory entries, packed as tightly as possible with no alignment constraints. Entries are added to the directory in order of file creation and are not sorted in any way.

Here is a description of the fields and a breakdown of the values in the five directories in this inode:

```
      Len (Bytes)      Field
          1            Length of file name (in bytes)
          2            Entry offset in non short form directory
          varies       Characters in file name
          1            File type
          4 or 8       Absolute inode address
```

```
Len    Offset     Name           Type     Inode
===    ======     ====           ====     =====
12     0x0060     01_smallfile   01       0x0417979d
10     0x0078     02_bigfile     01       0x0417979e
12     0x0090     03_smallfile   01       0x0417979f
10     0x00a8     04_bigfile     01       0x0417a154
12     0x00c0     05_smallfile   01       0x0417a155
```

First we have a single byte for the file name length in bytes. Like other Unix file systems, there is a 255 character file name limit.

The next two bytes are based on the byte offset the directory entry would have if it were a normal XFS directory entry and not packed into a short form directory in the inode. In a normal directory block, directory entries are 64-bit aligned and start at byte offset 96 (0x60) following the directory header and “.” and “..” entries. The directory entries here are all 18 or 20 bytes long, which means they would consume 24 bytes (0x18) in a normal directory block. Using a consistent numbering scheme for the offset makes it easier to write code that iterates through directory entries, even though the offsets don’t match the actual offset of each directory entry in the short form style.

Next we have the characters in the file name followed by a single byte for the file type. The file type is included in the directory entry so that commands like “ls -F” don’t have to open each inode to get the file type information. The file type values in the directory entry do not use the same number scheme as the file type in the inode. Here are the expected values for directory entries:

```
1   Regular file
2   Directory
3   Character special device
4   Block special device
5   FIFO
6   Socket
7   Symlink
```

Finally there is a field to hold the inode associated with the file name. In our example, these inode entries are 32 bits. 64-bit inode fields will be used if the directory header indicates they are needed.

Deleting a File

When a file is deleted from (or added to) a directory, the mtime and ctime in the directory’s inode core are updated. The directory file size changes (bytes 56-63). The CRC32 checksum and the logfile sequence number fields are updated.

In the data fork, all directory entries after the deleted entry are shifted downwards, completely overwriting the deleted entry. Here’s what the directory entries look like after “03_smallfile”– the third entry in the original directory– is deleted:

```
0x0B0 04 00 04 15 9F A1 0C 00 60 30 31 5F 73 6D 61 6C  .....¡ .`01_smal
0x0C0 6C 66 69 6C 65 01 04 17 97 9D 0A 00 78 30 32 5F  lfile..... .x02_
0x0D0 62 69 67 66 69 6C 65 01 04 17 97 9E 0A 00 A8 30  bigfile..... .¨0
0x0E0 34 5F 62 69 67 66 69 6C 65 01 04 17 A1 54 0C 00  4_bigfile...¡T .
0x0F0 C0 30 35 5F 73 6D 61 6C 6C 66 69 6C 65 01 04 17  À05_smallfile...
0x100 A1 55 0C 00 C0 30 35 5F 73 6D 61 6C 6C 66 69 6C  ¡U .À05_smallfil
0x110 65 01 04 17 A1 55 00 00 00 00 00 00 00 00 00 00  e...¡U..........
```

The four remaining directory entries are highlighted above. However, after those entries you can clearly see the residue of the entry for “05_smallfile” from the original directory. So as short-form directories shrink, they leave behind entries in the unused “inode slack”. In this case the residue is for a file entry that still exists in the directory, but it’s possible that we might get residue of entries deleted from the end of the directory list.

When Directories Grow Up

Another place you can see short form directory residue is when the directory gets large enough that it needs to move out to blocks on disk. I created a sample directory that initially had five files and confirmed that it was being stored as a short form directory in the inode. Then I added 45 more files to the directory, which made a short form directory impossible. Here's what the first part of the inode looks like after these two operations:

```
0x000 49 4E 41 ED 03 02 00 00 00 00 00 00 00 00 00 00  INAí............
0x010 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x020 5A EF 30 10 15 0C E1 74 5A EF 30 06 11 14 CB 44  Zï0.. átZï0...ËD
0x030 5A EF 30 06 11 14 CB 44 00 00 00 00 00 00 10 00  Zï0...ËD........
0x040 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01  ................
0x050 00 00 23 01 00 00 00 00 00 00 00 00 C8 E9 E2 AE  ..#.........Èéâ®
0x060 FF FF FF FF DF B0 11 11 00 00 00 00 00 00 00 01  ÿÿÿÿß°..........
0x070 00 00 00 20 00 00 62 EB 00 00 00 00 00 00 00 00  ... ..bë........
0x080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x090 5A EF 2F D8 32 2C 11 95 00 00 00 00 00 0B E4 D7  Zï/Ø2,....... äx
0x0A0 E5 6C 3B 41 CA 03 4B 41 B1 5C DD 60 9C B7 DA 71  ål;AÊ.KA±\Ý`.·Úq
0x0B0 00 00 00 00 00 00 00 00 00 00 00 2E 5B 40 00 01  ............[@..
0x0C0 6C 66 69 6C 65 01 00 0B E4 D8 0A 00 78 30 32 5F  lfile.. äØ .x02_
0x0D0 62 69 67 66 69 6C 65 01 00 0B E4 D9 0C 00 90 30  bigfile.. äÙ ..0
0x0E0 33 5F 73 6D 61 6C 6C 66 69 6C 65 01 00 0B E4 DA  3_smallfile.. äÚ
0x0F0 0A 00 A8 30 34 5F 62 69 67 66 69 6C 65 01 00 0B   .¨04_bigfile..
0x100 E4 DB 0C 00 C0 30 35 5F 73 6D 61 6C 6C 66 69 6C  äÛ .À05_smallfil
0x110 65 01 00 0B E4 DC 00 00 00 00 00 00 00 00 00 00  e.. äÜ..........
0x120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The data fork type (byte 5) is 2, meaning an extent list after the inode core, giving the location of the directory content on disk. You can see the extent highlighted starting at byte offset 176 (0xb0). But immediately after that extent you can see the residue of the original short-form directory.

The format of directories changes significantly when directory entries move out into disk blocks. In our next installment we will examine the structures in these larger directories.

*Posted in Forensics**Tagged File Systems, Linux, XFS*

Published by Hal Pomeranz

Independent Computer Forensics and Information Security consultant. Expert Witness. Trainer. *View all posts by Hal Pomeranz*