XFS (Part 5) – Multi-Block Directories

Posted on June 6, 2018June 11, 2018 by Hal Pomeranz

Life gets more interesting when directories get large enough to occupy multiple blocks. Let's take a look at my /etc directory:

```
[root@localhost hal]# ls -lid /etc
67146849 drwxr-xr-x. 141 root root 8192 May 26 20:37 /etc
```

The file size is 8192 bytes, or two 4K blocks.

Now we'll use xfs_db to get more information:

```
xfs_db> inode 67146849
xfs_db> print
[...]
core.size = 8192
core.nblocks = 3
core.extsize = 0
core.nextents = 3
[...]
u3.bmx[0-2] = [startoff,startblock,blockcount,extentflag]
0:[0,8393423,1,0]
1:[1,8397532,1,0]
2:[8388608,8394766,1,0]
[...]
```

I've removed much of the output here to make things more readable. The directory file is fragmented, requiring multiple single-block extents, which is common for directories in XFS. The directory would start as a single block. Eventually enough files will be added to the directory that it needs more than one block to hold all the file entries. But by this time, the blocks immediately following the original directory block have been consumed– often by the files which make up the content of the directory. When the directory needs to grow, it typically has to fragment.

What is really interesting about multi-block directories in XFS is that they are sparse files. Looking at the list of extents at the end of the xfs_db output, we see that the first two blocks are at logical block offsets 0 and 1, but the third block is at logical block offset 8388608. What the heck is going on here?

If you recall from our discussion of block directories in the last installment (https://righteousit.wordpress.com/2018/05/31/xfs-part-4-block-directories/), XFS directories have a hash lookup table at the end for faster searching. When a directory consumes multiple blocks, the hash lookup table and "tail record" move into their own block. For consistency, XFS places this information at logical offset XFS_DIR2_LEAF_OFFSET, which is currently set to 32GB. 32GB divided by our 4K block size gives a logical block offset of 8388608.

From a file size perspective, we can see that xfs_db agrees with our earlier ls output, saying the directory is 8192 bytes. However, the xfs_db output clearly shows that the directory consumes three blocks, which should give it a file size of 3*4096 = 12288 bytes. Based on my testing, the directory "size" in XFS only counts the blocks that contain directory entries.

We can use xfs_db to examine the directory data blocks in more detail:

```
xfs_db> addr u3.bmx[0].startblock
xfs_db> print
dhdr.hdr.magic = 0x58444433 ("XDD3")
dhdr.hdr.crc = 0xe3a7892d (correct)
dhdr.hdr.bno = 38872696
dhdr.hdr.lsn = 0x2200007442
dhdr.hdr.uuid = e56c3b41-ca03-4b41-b15c-dd609cb7da71
dhdr.hdr.owner = 67146849
dhdr.bestfree[0].offset = 0x220
dhdr.bestfree[0].length = 0x8
dhdr.bestfree[1].offset = 0x258
dhdr.bestfree[1].length = 0x8
dhdr.bestfree[2].offset = 0x368
dhdr.bestfree[2].length = 0x8
du[0].inumber = 67146849
du[0].namelen = 1
du[0].name = "."
du[0].filetype = 2
du[0].tag = 0x40
du[1].inumber = 64
du[1].namelen = 2
du[1].name = ".."
du[1].filetype = 2
du[1].tag = 0x50
du[2].inumber = 34100330
du[2].namelen = 5
du[2].name = "fstab"
du[2].filetype = 1
du[2].tag = 0x60
du[3].inumber = 67146851
du[3].namelen = 8
du[3].name = "crypttab"
[...]
```

I'm using the addr command in xfs_db to select the startblock value from the first extent in the array (the zero element of the array).

The beginning of this first data block is nearly identical to the block directories we looked at previously. The only difference is that single block directories have a magic number "XDB3", while data blocks in multi-block directories use "XDD3" as we see here. Remember that the value that xfs_db lobels dhdr.hdr.bno is actually the sector offset to this block and not the block number.

Let's look at the next data block:

```
xfs_db> inode 67146849
xfs_db> addr u3.bmx[1].startblock
xfs_db> print
dhdr.hdr.magic = 0x58444433 ("XDD3")
dhdr.hdr.crc = 0xa0dba9dc (correct)
dhdr.hdr.bno = 38905568
dhdr.hdr.lsn = 0x2200007442
dhdr.hdr.uuid = e56c3b41-ca03-4b41-b15c-dd609cb7da71
dhdr.hdr.owner = 67146849
dhdr.bestfree[0].offset = 0xad8
dhdr.bestfree[0].length = 0x20
dhdr.bestfree[1].offset = 0xc18
dhdr.bestfree[1].length = 0x20
dhdr.bestfree[2].offset = 0xd78
dhdr.bestfree[2].length = 0x20
du[0].inumber = 67637117
du[0].namelen = 10
du[0].name = "machine-id"
du[0].filetype = 1
du[0].tag = 0x40
du[1].inumber = 67146855
du[1].namelen = 9
du[1].name = "localtime"
[...]
```

Again we see the same header information. Note that each data block has it's own "free space" array, tracking available space in that data block.

Finally, we have the block containing the hash lookup table and tail record. We could use xfs_db to decode this block, but it turns out that there are some interesting internal structures to see here. Here's the hex editor view of the start of the block:

```
0x0000  00 00 00 00 00 00 00 00 3D F1 00 00 EF 65 44 61  ........=ñ..ïeDa
0x0010  00 00 00 00 02 51 50 70 00 00 00 22 00 00 87 20  .....QPp..."...
0x0020  E5 6C 3B 41 CA 03 4B 41 B1 5C DD 60 9C B7 DA 71  ål;AÊ.KA±\Ý`.·Úq
0x0030  00 00 00 00 04 00 94 61 01 26 00 01 00 00 00 00  .......a.&......
0x0040  00 00 00 2E 00 00 00 08 00 00 17 2E 00 00 00 0A  ................
0x0050  00 00 34 70 00 00 00 A4 00 00 38 6D 00 00 01 23  ..4p...¤..8m...#
0x0060  00 16 18 B1 00 00 01 1B 00 19 F2 6D 00 00 01 4D  ...±......òm...M
0x0070  00 19 F9 F3 00 00 01 F5 00 1A BB 6D 00 00 00 A2  ..ùó...õ..»m...¢
0x0080  00 1B 39 ED 00 00 00 3C 00 1B 3B 6D 00 00 00 52  ..9í...<..;m...R
0x0090  00 1B BA 70 00 00 03 A4 00 1B F8 74 00 00 01 21  ..ºp...¤..øt...!
0x00A0  00 1C 31 F0 00 00 01 EF 00 1C 35 E9 00 00 00 1D  . 1ð...ï. 5é...
0x00B0  00 1C 38 70 00 00 00 33 00 1C B0 F3 00 00 03 E7  . 8p...3. °ó...ç
0x00C0  00 1C B8 63 00 00 01 52 00 1C B8 6D 00 00 00 1F  . ¸c...R. ¸m...
0x00D0  00 1C F1 EC 00 00 02 71 00 1C F9 E8 00 00 00 64  . ñì...q. ùè...d
```

:

```
0-3     Forward link                     0
4-7     Backward link                    0
8-9     Magic number                     0x3df1
10-11   Padding                          zeroed
12-15   CRC32                            0xef654461

16-23   Sector offset                    38883440
24-31   Log seq number last update       0x2200008720
32-47   UUID                             e56c3b41-...-dd609cb7da71
48-55   Inode number                     67146849

56-57   Number of entries                0x0126 = 294
58-59   Unused entries                   1
60-63   Padding for alignment            zeroed
```

The “forward” and “backward” links would come into play if this were a multi-node B+Tree data structure rather than a single block. Unlike previous magic number values, the magic value here (0x3df1) does not correspond to printable ASCII characters.

After the typical XFS header information, there is a two-byte value tracking the number of entries in the directory, and therefore the number of entries in the hash lookup table that follows. The next two bytes tell us that there is one unused entry– typically a record for a deleted file.

We find this unused record near the end of the hash lookup array. The entry starting at block offset 0x840 has an offset value of zero, indicating the entry is unused:

```
0x0840 D9 C0 20 37 00 00 00 00 DB 9A A9 AF 00 00 03 41  ÙÀ 7....Û.©¯...A
0x0850 DB BF AC F8 00 00 03 CB DC 39 F4 E5 00 00 00 61  Û¿¬ø...ËÜ9ôå...a
0x0860 DC 3D B2 E8 00 00 00 B0 DD 1C C9 A6 00 00 02 82  Ü=²è...°Ý É¦....
0x0870 DD 30 80 E2 00 00 01 73 DD D2 24 F9 00 00 01 32  Ý0.â...sÝÒ$ù...2
0x0880 DE AA B4 CF 00 00 02 D9 E0 A1 02 31 00 00 00 21  Þª´Ï...Ùà¡.1...!
0x0890 E4 79 AD CC 00 00 03 F0 E6 DF 21 37 00 00 02 37  äy-Ì...ðæß!7...7
0x08A0 EA A3 91 E6 00 00 03 ED EC 88 20 0C 00 00 01 14  ê£.æ...íì. ....
0x08B0 EC 98 F0 DB 00 00 02 F5 EC 9A 78 E4 00 00 03 B3  ì.ðÛ...õì.xä...³
0x08C0 EC 9C 48 FC 00 00 00 9B EC A6 6C FC 00 00 03 4F  ì.Hü....ì¦lü...O
0x08D0 EC BE 7D 1F 00 00 03 32 EC DA 3A 2D 00 00 02 4E  ì¾} ...2ìÚ:-...N
0x08E0 ED 3D 14 2A 00 00 01 67 ED 46 F3 DE 00 00 00 E0  í=.*...gíFóÞ...à
0x08F0 ED 7C E8 52 00 00 00 E7 EE 72 E0 B6 00 00 00 BB  í|èR...çîrà¶...»
0x0900 EE 79 3B BE 00 00 00 DD F3 6D 84 8D 00 00 00 35  îy;¾...Ýóm.....5
0x0910 F3 F3 C9 CE 00 00 03 8A F5 18 65 52 00 00 02 E1  óóÉÎ....õ.eR...á
0x0920 F5 1F 21 44 00 00 03 6E F5 EF 70 A6 00 00 02 11  õ !D...nõïp¦....
0x0930 F7 23 3D 77 00 00 01 AC F9 7C F3 F5 00 00 00 D9  ÷#=w...¬ù|óõ...Ù
0x0940 FC 7A A5 CB 00 00 03 68 FC DB B5 F3 00 00 00 E4  üz¥Ë...hüÛµó...ä
0x0950 FD 35 F5 F3 00 00 03 0C FD DC BF 74 00 00 02 52  ý5õó... ýÜ¿t...R
0x0960 FE 1D 14 E2 00 00 01 5A FE 6E F1 24 00 00 01 CB  þ .â...Zþnñ$...Ë
0x0970 00 29 01 00 07 1B 04 73 65 6C 69 6E 75 78 73 79  .).....selinuxsy
0x0980 73 74 65 6D 5F 75 3A 6F 62 6A 65 63 74 5F 72 3A  stem_u:object_r:
0x0990 75 73 72 5F 74 3A 73 30 00 00 00 00 00 00 00 00  usr_t:s0........
0x09A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Interestingly, right after the end of the hash lookup array, we see what appears to be the extended attribute information from an inode. This is apparently residual data left over from an earlier use of the block.

At the end of the block is data which tracks free space in the directory:

```
0x0FA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0FC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0FD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x0FF0 00 00 00 00 00 00 00 08 00 08 00 20 00 00 00 02   ........... ....
```

The last four bytes in the block are the number of blocks containing directory entries– two in this case. Preceding those four bytes is a "best free" array that tracks the length of the largest chunk of free space in each block. You will notice that the array values here correspond to the dhdr.bestfree[0].length values for each block in the xfs_db output above. When new directory entries are added, this array helps the file system locate the best spot to place the new entry.

We see the two bytes immediately before the "best free" array are identical to the first entry in the array. Did the /etc directory once consume three blocks and later shrink back to two? Based on limited testing, this appears to be the case. Unlike directories in traditional Unix file systems, which never shrink once blocks have been allocated, XFS directories will grow and shrink dynamically as needed.

So far we've looked at the three most common directory types in XFS: small "short form" directories stored in the inode, single block directories, and in this case a multi-block directories tracked with an extent array in the inode. In rare cases, when the directory is very large and very fragmented, the extent array in the inode is insufficient. In these cases, XFS uses a B+Tree to track the extent information. We will examine this scenario in the next installment.

Posted in Forensics, Uncategorized 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*