Righteous IT *Join the crusade!*

XFS (Part 2) – Inodes

Posted on May 23, 2018January 11, 2022 by Hal Pomeranz

Part 1 (https://righteousit.wordpress.com/2018/05/21/xfs-part-1-superblock/) of this series was a quick introduction to XFS, the XFS superblock, and the unique Allocation Group (AG) based addressing scheme used in the file system. With this information, we were able to extract an inode from its physical location on disk

In this installment, we will look at the structure of the XFS inode. Since we will want to see what remains in the inode after a file is deleted, I'm going to create a small file for testing purposes:

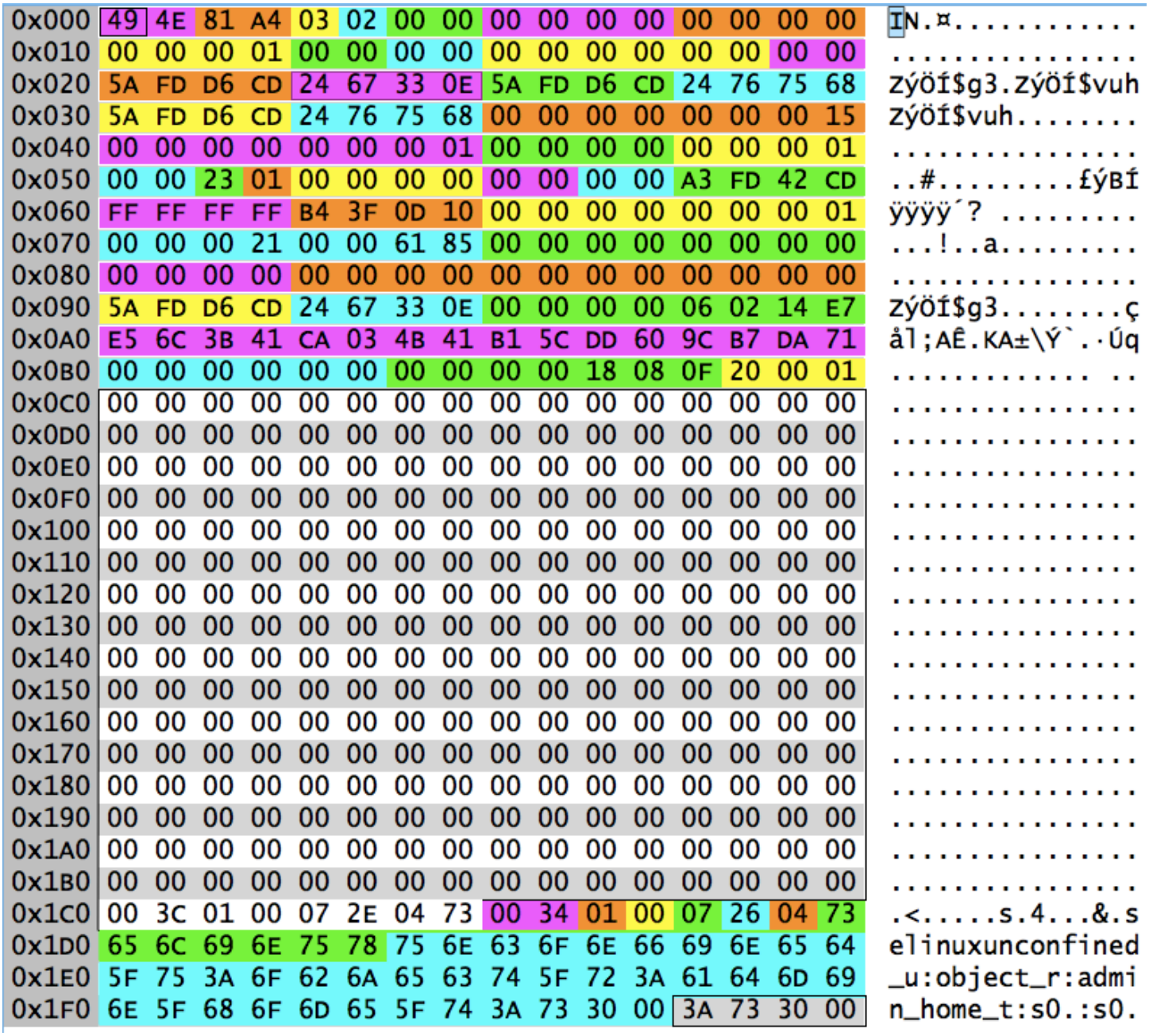
```
[root@localhost ~]# echo This is a small file >testfile
[root@localhost ~]# ls -i testfile
100799719 testfile
```

To save time, we'll use the xfs_db program to convert that inode address into the values we need to extract the inode from its physical location on disk. Then we'll use dd to extract the inode as we did in Part 1 (https://righteousit.wordpress.com/2018/05/21/xfs-part-1-superblock/).

```
[root@localhost ~]# xfs_db -r /dev/mapper/centos-root
xfs_db> convert inode 100799719 agno
0x3 (3)
xfs_db> convert inode 100799719 agblock
0x429c (17052)
xfs_db> convert inode 100799719 offset
0x7 (7)
xfs_db> ^D
[root@localhost ~]# dd if=/dev/mapper/centos-root bs=4096 \
                        skip=$((3*2427136 + 17052)) count=1 |
                    dd bs=512 skip=7 count=1 >/home/hal/testfile-inode
```

Looking at the Inode

We can now view the inode in our trusty hex editor:



0x000	49	4E	81	A4	03	02	00	00	00	00	00	00	00	00	00	00
0x010	00	00	00	01	00	00	00	00	00	00	00	00	00	00	00	00
0x020	5A	FD	D6	CD	24	67	33	0E	5A	FD	D6	CD	24	76	75	68
0x030	5A	FD	D6	CD	24	76	75	68	00	00	00	00	00	00	00	15
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	A3	FD	42	CD
0x060	FF	FF	FF	FF	B4	3F	0D	10	00	00	00	00	00	00	00	01
0x070	00	00	00	21	00	00	61	85	00	00	00	00	00	00	00	00
0x080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x090	5A	FD	D6	CD	24	67	33	0E	00	00	00	00	06	02	14	E7
0x0A0	E5	6C	3B	41	CA	03	4B	41	B1	5C	DD	60	9C	B7	DA	71
0x0B0	00	00	00	00	00	00	00	00	00	00	18	08	0F	20	00	01
0x0C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
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	3C	01	00	07	2E	04	73	00	34	01	00	07	26	04	73
0x1D0	65	6C	69	6E	75	78	75	6E	63	6F	6E	66	69	6E	65	64
0x1E0	5F	75	3A	6F	62	6A	65	63	74	5F	72	3A	61	64	6D	69
0x1F0	6E	5F	68	6F	6D	65	5F	74	3A	73	30	00	3A	73	30	00

XFS v5 inodes start with a 176 byte “inode core” structure:

```
0-1      Magic number                                "IN"
2-3      File type and mode bits (see below)         1000 000 110 100 100
4        Version (v5 file system uses v3 inodes)     3
5        Data fork type flag (see below)             2
6-7      v1 inode numlinks field (not used in v3)    zeroed
8-11     File owner UID                              0 (root)
12-15    File GID                                    0 (root)

16-19    v2+ number of links                         1
20-21    Project ID (low)                            0
22-23    Project ID (high)                           0
24-29    Padding (must be zero)                      0
30-31    Increment on flush                          0

32-35    atime epoch seconds                         0x5afdd6cd
36-39    atime nanoseconds                           0x2467330e
40-43    mtime epoch seconds                         0x5afdd6cd
44-47    mtime nanoseconds                           0x24767568

48-51    ctime epoch seconds                         0x5afd d6cd
52-55    ctime nanoseconds                           0x2476 7568
56-63    File (data fork) size                       0x15 = 21

64-71    Number of blocks in data fork               1
72-75    Extent size hint                            zeroed
76-79    Number of data extents used                 1

80-81    Number of extended attribute extents        0
82       Inode offset to xattr (8 byte multiples)    0x23 = 35 * 8 = 280
83       Extended attribute type flag (see below)    1
84-87    DMAPI event mask                            0
88-89    DMAPI state                                 0
90-91    Flags                                       0 (none set)
92-95    Generation number                           0xa3fd42cd

96-99    Next unlinked ptr (if inode unlinked)      -1 (NULL in XFS)

/* v3 inodes (v5 file system) have the following fields */
100-103  CRC32 checksum for inode                    0xb43f0d10
104-111  Number of changes to attributes             1

112-119  Log sequence number of last update          0x2100006185
120-127  Extended flags                              0 (none set)

128-131  Copy on write extent size hint             0
132-143  Padding for future use                     0

144-147  btime epoch seconds                        0x5afdd6cd
148-151  btime nanoseconds                          0x2467330e
152-159  inode number of this inode                 0x60214e7 = 100799719
```

```
160-175  UUID                                  e56c3b41-...-dd609cb7da7
```

XFS inodes start with the 2 byte magic number value “IN”. Inodes also have a CRC32 checksum (bytes 100-103) to help detect corruption. The inode includes its own absolute inode number (bytes 152-159) and the file system UUID (bytes 160-175), which should match the UUID value from the superblock. Whenever the inode is updated, bytes 112-119 track the “logfile sequence number” (LSN) of the journal entry for the update. The inode format has changed across different versions of the XFS file system, so refer to the inode version in byte 4 before decoding the inode. XFS v5 uses v3 inodes.

The size of the file (in bytes) is a 64-bit value in bytes 56-63. The original XFS inode tracked the number of links as a 16-bit value (bytes 6-7), which is no longer used. Number of links is now tracked as a 32-bit value found in bytes 16-19.

Timestamps include both a 32-bit “Unix epoch” style seconds field and a 32-bit nanosecond resolution fractional seconds field. The three classic Unix timestamps– atime, mtime, ctime– are found in bytes 32-55 of the inode. File creation time (btime) was only added in XFS v5, so that timestamp resides in bytes 144-151 in the upper portion of the inode core.

File ownership and permissions are tracked as in earlier Unix file systems. There are 32-bit file owner (bytes 8-11) and group owner (bytes 12-15) fields. File type and permissions are stored in a packed 16-bit structure. The low 12 bits are the standard Unix permissions bits, and the upper four bits are used for the file type.

The file type nibble will be one of the following values:

```
8   Regular file
4   Directory
2   Character special device
6   Block special device
1   FIFO
C   Socket
A   Symlink
```

The 12 permissions bits are grouped into four groups of 3 bits, and are often written in octal notation– in our case we have 0644. The first group of three represents the “special” bit flags: set-UID, set-GID, and “sticky” (none of these are set for our test file). The remaining three groups represent “read” (r), “write” (w), and “execute” (x) permissions for three categories. The first set of bits applies to the file owner, the second to members of the Unix group that owns the file, and the last group for everybody else. The permissions on our test file are 644 or 110 100 100 aka rw-r–r–. In other words, read and write access for the file owner, and read only access for group members and for all other users on the system.

The remaining space after the 176 bytes of inode core is used to track the data blocks associated with the file (the “data fork” of the file) and any extended attributes that may be set. There are multiple ways in which data and attributes may be stored– locally resident within the inode, in a series of extents, or in a more complex B+Tree indexed structure. The data fork type flag in byte 5 and the extended attribute type flag in byte 83 document how this information is organized. The possible values for these fields are:

```
0   Special device file (data type only)
1   Data is resident ("local") in the inode
2   Array of extent structures follows
3   B+Tree root follows
```

Currently XFS only uses resident or “local” storage for extended attributes and small directories. There is a proposal to allow small files to be stored in the inode (similar to NTFS), but this is still under development. The data fork for our small test file is type 2– an array of extent structures. The extended attributes are type 1, meaning they are stored locally in the inode.

The data fork starts at byte 176, immediately after the inode core. The start of the extended attribute data is found at an offset from the end of the inode core. This offset is byte 82 of the inode core, and the units are multiples of 8 bytes. In our sample inode, the offset value is 0x23 or 35. Multiplying by 8 gives a byte offset of 280 from the end of the inode core, or 176+280=456 bytes from the beginning of the inode.

Extent Arrays

The most common storage option for file content in XFS is data fork type 2– an array of 16 byte extent structures starting immediately after the inode core. Bytes 76-79 indicate how many extent structures are in the array. Our file is not fragmented, so there is only a single extent structure in the inode.

Theoretically, the 336 bytes following the inode core could hold 21 extent structures, assuming no extended attribute data. If the inode cannot hold all of the extent information (an extremely fragmented file), then the data fork in the inode becomes the root of a B+Tree (data fork type 3) for tracking extent information. We will see an example of this in a later installment in this series.

The challenging thing about XFS extent structures is that they are not byte aligned. They contain four fields as follows:

- Flag (1 bit) – Set if extent is preallocated but not yet written, zero otherwise
- Logical offset (54 bits) – Logical offset from the start of the file
- Starting block (52 bits) – Absolute block address of the start of the extent
- Length (21 bits) – Number of blocks in the extent

If you think this makes manually decoding XFS extent information challenging, you’d be correct. Let’s break the extent structure down into individual bits in order to make decoding a bit easier. The extent starts at byte offset 176 (0xb0), and I’ll use a little command-line magic to see the bits:

```
[root@localhost ~]# xxd -b -c 4 /home/hal/testfile-inode |
                      grep -A3 0b0:
00000b0: 00000000 00000000 00000000 00000000  ....
00000b4: 00000000 00000000 00000000 00000000  ....
00000b8: 00000000 00000000 00011000 00001000  ....
00000bc: 00001111 00100000 00000000 00000001  . ..

Flag bit (1 bit): 0
logical offset (54 bits): 0
absolute start block (52 bits):
    0 00000000 00000000 00000000 00011000 00001000 00001111 001

    0000 0000 0000 0000 0000 0000 0000 1100 0000 0100 0000 0111 1001
      0    0    0    0    0    0    0    C    0    4    0    7    9

    block 0xC04079 aka relative block 0x4079 (16505) in AG 3

block count (21 bits): 1
```

Let’s check and see if we decoded the structure correctly:

```
[root@localhost ~]# dd if=/dev/mapper/centos-root bs=4096
                      skip=$((3*2427136 + 16505)) count=1 | xxd
0000000: 5468 6973 2069 7320 6120 736d 616c 6c20  This is a small
0000010: 6669 6c65 0a00 0000 0000 0000 0000 0000  file............
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
[... all zeroes to end ...]
```

Looks like we got it right. Note that XFS null fills file slack space, which is typical for Unix file systems.

Extended Attributes

XFS allows arbitrary extended attributes to be added to the file. Attributes are simply name, value pairs. There is a 255 byte limit on the size of any attribute name or value. You can set or view attributes from the command line with the “attr” command.

If the amount of attribute data is small, extended attributes will be stored in the inode, just as they are in our sample file. Large amounts of attribute information may need to be stored in data blocks on disk, in which case the attribute data is tracked using extents just like the data fork.

As we discussed above, resident attribute information starts at a specific byte offset from the end of the inode core. In our sample file the offset is 280 bytes from the end of the inode core or 456 bytes (280 + 176) from the start of the inode.

Attributes start with a four byte header:

```
456-457  Length of attributes                0x34 = 52
458      Number of attributes to follow      1
459      Padding for alignment               0
```

The length field unit is bytes and includes the 4 byte header. Our sample file only contains a single attribute.

Each attribute structure is variable length, to allow attributes to be packed as tightly as possible. Each attribute structure starts with a single byte for the name length, then a byte for the value length, and a flag byte. The rest of the attribute structure is the name followed by the value, with no null terminators or padding for byte alignment.

Breaking down the single attribute we have in our sample inode, we see:

```
460      Length of name                      7
461      Length of value                     0x26 = 38
462      Flags                               4
463-469  Attribute name                      selinux
470-507  Attribute value                     unconfined_u:...
```

This attribute holds the SELinux context on our file, “unconfined_u:object_r:admin_home_t:s0”. While extended attribute values are not required to be null-terminated, SELinux expects it’s context labels to have null terminators. So the 38 byte value length is 37 printable characters and a null.

The flags field is designed to control access to the attribute information. The flags byte is defined as a bit mask, but only four values appear to be used currently:

```
  128   Attribute is being updated
    4   "Secure" - attribute may be viewed by all but only set by root
    2   "Trusted" - attribute may only be viewed and set by root
    0   No restrictions
```

The Inode After Deletion

When a file is deleted, changes are limited to a small number of fields in the inode core:

- The 2 byte file type and permissions field is zeroed
- Link count, file size, number of blocks, and number of extents are zeroed
- ctime is set to the time the file was deleted
- The offset to the extended attributes is zeroed
- The data fork and extended attribute type bytes are set to 2, which would normally mean an extent array
- The “Generation number” field (inode bytes 92-95) is incremented–more testing is required, but it appears this field may be a usage count for the inode
- The CRC32 checksum and the LSN are updated

No other data in the inode changes. So while the number of extents value is zeroed and so is the offset to the start of the extended attributes, the actual extent and attribute data remains in the inode.

This means it should be straightforward to recover the original file by parsing whatever extent data exists starting at inode offset 176. The XFS FAQ (http://xfs.org/index.php/XFS_FAQ#Q:_Does_the_filesystem_have_an_undelete_capability.3F) points to two Open Source projects that appear to use this idea to recover deleted files, and a little Google searching turns up several commercial tools that claim to do XFS file recovery:

- xfsr (https://github.com/salviati/xfsr) (Open Source)
- xfs_irecover (http://inai.de/projects/hxtools/) (Open Source)
- ReclaiMe (https://www.reclaime.com/library/xfs-recovery.aspx)
- UFS Explorer (http://www.ufsexplorer.com/download_pro.php)
- XFS Data Recovery (https://www.stellarinfo.com/disk-recovery/xfs-recovery.php)

I have not had the opportunity to test any of these tools.

In limited testing it also appears that the data fork and the extended attribute information are not zeroed when the inode is reused. This means there is the possibility of finding remnants of data from a previous file in the unused or “slack” space in the inode.

Using xfs_db to View Inodes

xfs_db allows you to quickly view the inode values, even for inodes that are currently unallocated:

```
[root@localhost ~]# xfs_db -r /dev/mapper/centos-root
xfs_db> inode 100799719
xfs_db> print
core.magic = 0x494e
core.mode = 0
core.version = 3
core.format = 2 (extents)
core.nlinkv2 = 0
core.onlink = 0
core.projid_lo = 0
core.projid_hi = 0
core.uid = 0
core.gid = 0
core.flushiter = 0
core.atime.sec = Thu May 17 16:41:15 2018
core.atime.nsec = 821506703
core.mtime.sec = Thu May 17 16:41:15 2018
core.mtime.nsec = 821506703
core.ctime.sec = Thu May 17 22:10:07 2018
core.ctime.nsec = 163429238
[... additional output not shown...]
```

xfs_db even converts the timestamps for you, so that’s a win.

What's Next?

XFS does not store file name information in the inode, which is pretty typical for Unix file systems. The only place where file names exist is in directory entries. In our next installment (https://righteousit.wordpress.com/2018/05/25/xfs-part-3-short-form-directories/) we will begin to examine the different XFS directory types. Yes, it's complicated.
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*

1 thought on "XFS (Part 2) – Inodes"

1. **Week 21 – 2018 – This Week In 4n6** *May 27, 2018*
   […] The second looks "at the structure of the XFS inode." XFS (Part 2) – Inodes […]

Comments are closed.