XFS
Practical
Exercises

04 – Extended Attributes

Overview

Extended Attributes (EA's) are used for attaching structured meta information to a file or inode and below we will give an example using an ACL. In the lab we will look at how we store an ACL in an EA and how this EA can be stored in various forms on the disk in XFS. Finally, we will look at how the EA's compete for inode space with the data extents and how variable this can be when attr2 is turned on for an XFS filesystem.

Goals

- To understand what EA's are in XFS
- The interfaces to manipulate EA's
- Summary of the ondisk formats
- The competition for inode space between data extents and EA extents
- Becoming more familiar with xfs_db(8) for looking at inodes

Prerequisites

Some knowledge of what Extended Attributes (EA's) and Access Control Lists (ACL's) are would be beneficial.

Setup

- Need a scratch filesystem to play with as root

- In the examples we will refer to this device and mount point through these environment variables, for example:

```
export SCRATCH_MNT=/mnt/scratch
export SCRATCH_DEV=/dev/sda8
```

- acl and attr packages installed

- xfsprogs package installed

- xfstests/src/makeextents binary; the directory path for this is called $BINDIR

```
export BINDIR=/home/fred/src/xfs-cmds/xfstests/src
```

- For the attr2 exercises:
- o a kernel which supports XFS attr2 format
- o a mkfs.xfs which supports XFS attr2 format

Exercises

Exercise 1 – ACLs and EA Interface

We create an access ACL on a file, and look at how its value is stored in an EA and what the EA for it looks like on the disk. A posix access ACL on a file is a set of access permissions on a file, like the standard unix permissions, but gives a finer grain of control on who gets these permissions. As the point of this lab is about XFS EAs and not about ACLs, we will just create a simple ACL with user, group, other permissions and the mask entry.

The setfacl(1) and getfacl(1) commands are standard commands which were implemented by Andreas Gruenbacher and the chacl(1) command came originally from IRIX.

Setup

1. create filesystem and ACL's

```
# cd /
# mkdir $SCRATCH_MNT
# mkfs.xfs -f $SCRATCH_DEV
# mount $SCRATCH_DEV $SCRATCH_MNT

# cd $SCRATCH_MNT
# echo data1 > file1
# echo data2 > file2
# setfacl -m u::rw,g::rw-,o::r--,m::rwx file1
# chacl u::r--,g::---,o::---,m::rwx file2

# getfacl file2
# chacl -l file1
```

Exercise

2. List the extended attributes on the file:

```
# getfattr -d file1
```

3. This won't show much as we didn't specify the "user" namespace. This will show two extended attributes:

```
# getfattr -e hex -dm '.*' file1
```

One for the "system" namespace and one for the "trusted" namespace.

4. However, when we run the "attr" command, it only shows 1 EA which is what we'd expect since we only created one ACL on the file.

```
# attr -Rl file1
# attr -Rqg SGI_ACL_FILE file1 >ea_value
# od -x ea_value
```

The reason why getfattr shows 2 EAs is because the system.posix_acl_access is an EA XFS provides as an interface into the system and ACL routines, however, the trusted.SGI_ACL_FILE EA is the only one actually stored on the disk and is the same as would be stored on an IRIX XFS filesystem. The internal namespace for this XFS EA is actually "root" which is stored as a bit in the flags field (it doesn't actually store the namespace as a string in XFS).

In our case, we have 4 entries: u::rwx g::rw- o::r-- m::rwx

5. An XFS ACL is of the form:

```
<acl_count: int32>  <entry> <entry> <entry> ...
```

where:

```
<entry> = <tag: int32> <id: int32> <perm: uint16>
```

and

```
<tag> = USER_OBJ, GROUP_OBJ, OTHER
<id> = user id or group id if given one
<perm> = normal unix permissions like rwx
```

You can now try to match up a few of the fields of the SGI_ACL_FILE EA contents with the format of an ACL given above; you can see, for example, that there are 4 entries for the acl_count which are at the start of the EA value.

6. Identify the permissions from the od (octal) dump

Exercise 2 - ondisk form of the EA (ACL)

We will now see how this ACL and other EAs look in terms of the on disk structure using xfs_db.

1. Obtain the inode number for the file

```
# ls -i file1
# umount $SCRATCH_MNT
```

2. Use xfs_db to look at what is on the disk to see:

- Data format is extents (core.format = 2)

- Offset between data and EA fork = 15 * 8 = 120 bytes

- EA format is extents (core.aformat = 2)

```
# xfs_db -r /dev/sda8
xfs_db> inode inode_number
xfs_db> print
...
core.format = 2 (extents)
core.forkoff = 15
core.aformat = 2 (extents)
u.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,13,1,0]
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,12,1,0]
xfs_db> ablock 0
xfs_db> p
....stuff omitted...
nvlist[0].valuelen = 52
nvlist[0].namelen = 12
nvlist[0].name = "SGI_ACL_FILE"
nvlist[0].value =
"\000\000\000\004\000\000\000\001\377\377\377\377\000\006\000\000\000\000\000\
004\377\377\377\377\000\006\000\000\000\000\000\020\377\377\377\377\000\a\000\
000\000\000\000 \377\377\377\377\000\004\000\000"
```

3. Alternatively specifying the filesystem block directly:

```
xfs_db> fsb 12
xfs_db> type attr
xfs_db> p
```

Look at the file data while we are here:

```
xfs_db> dblock 0
xfs_db> p
000: 64617461 310a0000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
...etc...
xfs_db> type text
xfs_db> p
000:  64 61 74 61 31 0a 00 00 00 00 00 00 00 00 00 00  data1...........
010:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

i.e. the file1 data is "data1" just like we echo'ed in at the beginning.

So in this example with 1 data extent for its contents, the ACL EA was also in extent format. However, EAs can often fit inside the inode in what is called "shortform" format. However, in this case, the EA space starts from the fork offset in the inode:

```
core.forkoff = 15
```

which is 15 * 8 = 120 bytes into the lIteral space of the inode (which is the area after the core part of the inode). With the 51 bytes value and 12 byte name and the rest of the EA header, there was not enough room for the EA to be inside the inode.

Exercise 3 – attr2 filesystem

With the attr2 format used in the mkfs.xfs parameters, the forkoff is not set at a fixed place anymore, as was in traditional XFS. Instead, the fork offset is moved according to the requirements of the EAs and until it bumps into the data extents, always leaving enough room for a btree root data extent block.

Another alternative to increasing the chance of fitting the EA within the inode, is to increase the size of the inode at mkfs time (using "-i size=xxx").

Please refer to the attr2 diagram in the XFS Filesystem Structure document on Extended Attributes.

Setup

1. Recreate the filesystem with attr2 enabled

```
# cd /
# unmount $SCRATCH_MNT
# mkfs.xfs -fi attr=2 $SCRATCH_DEV
# mount $SCRATCH_DEV $SCRATCH_MNT
```

2. Create a file and give it an ACL

```
# cd $SCRATCH_DEV
# echo data1 > file1
# setfacl -m u::rw,g::rw-,o::r--,m::rwx file1
```

Exercise

3. Re-examine the inode and observe how the fork offset and aformat have changed comparedto the earlier exercises.

Exercise 4 – ondisk EA's in different formats

As we know, if the EA is small enough it can fit within the inode. If we have an EA with a big name or a big value, or lots of EA's then it will move out of the inode. As soon as we are using EA's, there is then less room in the inode for the data extents. This exercise explores the different data structures used ondisk for different sized extended attributes.

Setup

1. Recreate the filesystem

```
# cd /
# umount $SCRATCH_MNT
# mkfs.xfs -f $SCRATCH_DEV
# mount $SCRATCH_DEV $SCRATCH_MNT
```

Exercise

2. Add an attribute to a file

```
# cd $SCRATCH_MNT
# echo data1>file1
# setfattr -n user.name1 -v value1 file1
# getfattr -d file1
```

3. Examine the inode's format to see it is in short-form with a fork offset of 120 bytes:

```
# ls -I file1
# cd /
# umount $SCRATCH_MNT
# xfs_db -r -c "inode inode_number" -c "p a" -c "p core.forkoff" -c "p
core.aformat" $SCRATCH_DEV
```

4. A another EA, name2, but this time with a big value of 60K.

```
# cd $SCRATCH_MNT
# man bash | strings | dd bs=1024 count=60 of=file2
# ls -s file2
61440 file2
# attr -s name2 file1 < file2 >/ dev/null
```

5. Dump the inode and explore its structure

```
# ls -i file2
# cd /
# umount $SCRATCH_MNT
# xfs_ncheck $SCRATCH_DEV
# xfs_db -r $SCRATCH_DEV
xfs_db> inode inode_number
xfs_db> p
```

The EA is quite large and will not fit within the inode, instead it is in its own set of blocks

```
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,28,16,0]
```

6. The first block at 28 contains the main EA information and the remaining 15 blocks (15 * 4K = 60K) starting from block 29 contain the value for name2, in this case the bash(1) man page..

```
xfs_db> fsb 28
xfs_db> type attr
xfs_db> p
xfs_db> fsb 29
xfs_db> type text
xfs_db> p
```

7. Now add 1000 attributes to the file:

```
# for i in {1..1000}
> do
> attr -s name.$i -V value.$i file1
> done
# cd /
# umount /mnt/scratch
# xfs_db -r $SCRATCH_DEV
```

This time we now have the EA in btree form with the root within the inode and the actual EA data in the leaf blocks of the btree, similar to:

```
btree[0-11] = [hashval,before] 0:[0x55101e5a,16] 1:[0x55105dd8,24]
2:[0x55109c5b,25] 3:[0x55109fde,23] 4:[0x5510dfde,21] 5:[0x55111ed7,22]
6:[0x55115ed7,19] 7:[0x5511dd5a,20] 8:[0x55139d5a,18] 9:[0x5513dd5a,26]
10:[0xdcaa20bd,27] 11:[0xec3b72b4,17]
```

8. Examine one of the leafblocks (in this example block16):

```
xfs_db> ablock 16
xfs_db> p
```

Exercise 5 - attr2 and inode literal space competition

With attr2, there is more competition allowed for the literal space of the inode. To show this we will create a file with 1 EA and see how many data extents we can fit within an inode. Then we will try this all again with 24 Eas and now see how many data extents we can fit inline. At each stage, you can look at what the new forkoffset is set to.

Setup

1. Create a filesystem with 512 byte inodes and attr2

```
# mkfs.xfs -f -i "attr=2,size=512" $SCRATCH_DEV
# mount $SCRATCH_DEV $SCRATCH_MNT
```

Exercise

2. Create and examine a file with a single attribute

```
# cd $SCRATCH_MNT
# touch file
# setfattr -n user.name.1 -v value file
# xfs_bmap file
# ls -i file
# cd /
# umount $SCRATCH_MNT
# xfs_ncheck $SCRATCH_DEV
# xfs_db -r $SCRATCH_DEV -c 'inode inode_number' -c 'p'
```

We can see from this output that we have 1 EA and a fork offset of 47 which is equivalent to 376 bytes worth of space left over for data extents.

3. Let's now see how many extents we can fill up before they go out of line.

```
# mount $SCRATCH_DEV $SCRATCH_MNT
# cd $SCRATCH_MNT
# $BINDIR/makeextents -p -n 23 file
```

The "-p" option is to preserve the file and its extents – it won't create 23 more extents, rather it should create enough extents to have 23 extents in total.

```
# xfs_bmap file
# cd /
# umount $SCRATCH_MNT
# xfs_db -r $SCRATCH_DEV -c 'inode inode_number' -c 'p'
```

From this we can see that we can fit 23 extents inline.

4. Now you should try it with 24 extents. Still inline?

5. Clear all the data extents in the file by truncating it to zero.

```
# mount $SCRATCH_DEV $SCRATCH_MNT
# cd $SCRATCH_MNT
# >file
# xfs_bmap file
```

6. Now add 23 more EA's so that we have 24 EA's in total.

```
# for i in {2..24}
> do
> setfattr -n user.name.$i -v value file
> done
# getfattr -d file
```

7. Now unmount and look at the fork offset in xfs_db. It should be at about 7 which is equivalent to 56 bytes into the literal space. So in adding 23 EA's we have gone from 376 bytes down to 56 bytes.

8. How many extents we can fit in there before going out of line for the data extents? Try first creating 3 extents and then 4.

```
# mount $SCRATCH_DEV $SCRATCH_MNT
# cd $SCRATCH_MNT
# $BINDIR/makeextents -p -n 3 file
# xfs_bmap file
# cd /
# umount $SCRATCH_MNT
# xfs_db -r $SCRATCH_DEV -c 'inode inode_number' -c 'p'
```

In the answers section below is a table which lists the various fork offsets for EA's created in the same way we did above. It also shows how many data extents we can fit inline with that fork offset – although only 2 rows for this column have been filled in.

Questions

1. If I have a file with EA's in it which they are all in shortform format (i.e. they are all inside the inode), and I decide to replace one of those EA's by specifying the same name in a setfattr command but this time using a large value (e.g. 60K), what will happen to the format of the EA?

    Will it just mark the value for this EA as remote and put a pointer to a set of blocks for this value and leave all rest of the EA's in shortform? Or as well as marking it remote and storing the value remotely, will it change the EA format to the extent form?

2. Construct a table showing how many inline extents are possible as the number of extended attributes grow.

Answers

1. It will change to the extent form with 1 filesystem block for the EA names and values, and will mark the large value EA (60K) as remote and store its value in remote blocks. We did this in the lab :)

2. This table shows for various number of EA's named "name.xx" and value "value" on a 512 byte inode,what the fork offset will be and the maximum number of inline extents one can have (only a couple of these entries have been filled in).

Number of EA's	Fork Offset	Fork Offset bytes	Max # of inline extents
1	47	376	23
2	47	376	
4	44	352	
8	37	296	
16	22	176	
20	14	112	
21	12	96	
22	10	80	
23	9	72	
24	7	56	3
25	24 (extent format)	192	