XFS Practical Exercises

03 - Allocators

Overview

Goals

This lab demonstrates how inode numbers are sized, and how inodes and extents are allocated based on different constraints and mount options.

Prerequisites

Setup

Define what software (and versions) are required and how it needs to be configured.

Exercises

Exercise 1

This exercise demonstrates how many bits an inode requires to be located anywhere in a filesystem.

Setup

1. Use dd(1) to write a single (count=1) 4kB block of data (bs=4K) to an offset that's 256 * 1024 blocks (seek=256K) from the start of the file named loop. An offset of 256 * 1024 * 4kB = 1GB, so the resulting file is 1GB plus 4kB in length.

```
# dd if=/dev/zero of=loop bs=4K seek=256K count=1 > /dev/null 2>&1
```

2. The ls command confirms this, rounded up to the nearest 0.1 of a GB.

```
# ls -lh loop
```

Exercise

3. mkfs.xfs is used to write a file system to the loop file and shows that the geometry is using
   a. inodes that are 256 bytes in size
   b. there are 8 allocation groups
   c. block size is 4kB

```
# mkfs.xfs -d file=loop loop
```

4. Running xfs_db(8) to examine the superblock gives three relevant values for the formatting of inode numbers on this file system.
   a. 'agcount' value of 8 or 23 means that three bits will be needed to specify the AG number.
   b. 'inopblog' value of 4 means that there are 24 or 16 inodes per filesystem block. This makes sense since 16 256 byte inodes can fit in a 4kB filesystem block.
   c. 'agblklog' value of 15 indicates that each AG has 215 filesystem blocks in it. This is expected since the mkfs.xfs output reported an agsize value of 32768 blocks.

```
# xfs_db -f -c "sb 0" -c "p" ./loop | egrep 'agcount|inopblog|agblklog'
```

5. 3 + 4 + 15 = 22 bits of data are required to address an inode placed anywhere in this filesystem

Exercise 2

This exercise demonstrates the point at which 32 bits is no longer enough to locate an inode anywhere in the filesystem.

1. Show that 32 bits are required for a 1kB inode located in a 4TB filesystem.
2. Slightly increase the size of the file to increase the AG size by seeking to 31 blocks beyond 4TB. The increased AG size has increased the inode size to 33 bits, it is no longer possible to place a 32 bit inode anywhere in the filesystem.

Exercise 3

This lab demonstrates the inode rotor in action.

Setup

1. Firstly, a filesystem large enough to trigger inode32 rotor behavior is needed. This can be achieved on a small local disk by creating a large sparse file, writing a new filesystem onto it and mounting it in loopback mode.

```
> dd if=/dev/zero of=loop bs=4K seek=1G count=1 > /dev/null 2>&1

> ls -sh loop

4.1T loop

> mkfs.xfs -d file=loop loop

meta-data=loop                   isize=256    agcount=32, agsize=33554432 blks

         =                       sectsz=512   attr=0

data     =                       bsize=4096   blocks=1073741824, imaxpct=25

         =                       sunit=0      swidth=0 blks, unwritten=1

naming   =version 2              bsize=4096

log      =internal log           bsize=4096   blocks=32768, version=1

         =                       sectsz=512   sunit=0 blks

realtime =none                   extsz=4096   blocks=0, rtextents=0

> xfs_db -f -c "sb 0" -c "p" ./loop | egrep 'agcount|inopblog|agblklog'

agcount = 32

inopblog = 4

agblklog = 25

> sudo mkdir /mnt/loop

> sudo mount -o loop loop /mnt/loop

> df -h /mnt/loop

Filesystem            Size  Used Avail Use% Mounted on

/home/sjv/loop        4.0T  528K  4.0T   1% /mnt/loop

> sudo chmod 777 /mnt/loop

> cd /mnt/loop
```

Exercise

2. Next, ten files are created in a directory on the new filesystem and their inode numbers and data extent allocations are examined:

```
> dir=a

> mkdir $dir

> for file in `seq 0 9`; do xfs_mkfile 10m $dir/$file; done

> ls -is $dir/*

> for file in `seq 0 9`; do

> ag=`xfs_bmap -v $dir/$file | tail -1 | awk '{print $4}'`

> echo $dir/$file is in AG $ag

> done
```

3. Repeat step 2 with a different rotorstep value:

```
> sudo sysctl -w fs.xfs.rotorstep=3

> dir=b
```

Exercise 4

This exercise demonstrates the need for the filestreams allocator for a particular workload.

Setup

1. Create a file system with a small AG size:

```
# mkfs.xfs -d agsize=64m /dev/sdb7 > /dev/null

# mount /dev/sdb7 /test

# chmod 777 /test
```

2. Create ten 10MB files concurrently in two directories:

```
# cd /test

# mkdir a b

# for dir in a b; do

> for file in `seq 0 9`; do

> xfs_mkfile 10m $dir/$file

> done &

> done; wait 2>/dev/null
```

Exercise

3. Note that the all the inodes in the same directory are in the same AG:

```
# ls -sid * */*
```

4. What about the file data? Use xfs_bmap -v to examine the extents of each of the files:

```
# xfs_bmap -v a/*

# xfs_bmap -v b/*
```

5. Note that once the original AG was used, the files from both directories are interleaved in the next allocation group. Any read-ahead for files in directory a that assumes the files are contiguous on disk will start reading files from directory b instead.

Questions

1. In exercise 3, why do the new files start in allocation group 8?
2. In exercise 3, what can be observed regarding the inode numbers of the files in the two directories?
3. In exercise 4, the file inodes are created in the same allocation group as the parent directory, but the file extents are not. Why is this?

Answers

1. The first 8 allocation groups make up the first TB and are reserved for inodes.
2. The inodes for the second directory are put into a different allocation group than the first directory, still within the first TB.
3. It's important to consider the order in which events are occurring. The two bash processes writing files are calling xfs_mkfile, which starts by opening a file with the O_CREAT flag. At this point, XFS has no idea how large the file's data is going to be, so it dutifully creates a new inode for the file in the same AG as the parent directory. The call returns successfully and the system continues with its tasks. When XFS is asked write the file data a short time later, a new AG must be found for it because the inode’s AG is full. The result is a violation of the original goal to keep file data close to its inode on disk. In practice, because inodes are allocated in clusters on disk, a process that's reading back a stream is likely to cache all the inodes it needs with just one or two reads, so the disk seeking involved won't be as bad as it first seems.