xFS Block Zeroing Mechanism

Adam Sweeney

1.0 Introduction

This document describes the mechanisms used in xFS to ensure that we never return uninitialized data to anyone reading a file. This could potentially occur when:

- Blocks are allocated to a file, but before the data for those blocks are written to disk the system crashes.
- A file with fixed sized extents is written sparsely. In this case what would have been holes in the file are actually allocated space on disk, but the user never specifies any data to be written into those holes. It is the job of the file system to ensure that the user cannot tell the difference between these holes and regular holes which always return zeros when read.

The goal of this design is to avoid whenever possible actually writing zeros to the disk or memory when the user has specified data to be written there. Writing zeros is a pessimistic strategy which adds cpu and I/O overhead to the file writing process. The only times in which writing zeroes is acceptable are when the user does not initialize that portion of the file (case 2 above) or during file system recovery. In place of actually writing the zeros to the file, xFS will place operations in the log which indicate that during recovery we should zero the specified file blocks. This way we use the transaction mechanism to eliminate the overhead of block zeroing.

2.0 Background

In xFS we use delayed allocation to improve the contiguity of our file block allocations. Instead of deciding which blocks to associate with a range of a file when the user initially writes to that range, we instead only reserve space on the disk for the necessary number of blocks and delay the choice of which blocks until we have to actually flush the user's data to disk. This way we take advantage of the UNIX write behind mechanisms to gain a better idea of the eventual size of the file.

When a delayed write, delayed allocation buffer is finally pushed out it is sent to the xfs_strat_write() routine. There we allocate blocks for the file range covered by the delayed allocation buffer as well as any other delayed allocation blocks in the file which are contiguous to ours. The idea is to allocate all of the blocks as a single large extent.

At the point at which we convert a delayed allocation extent into a real one we are writing out a buffer which covers some subset of the allocated extent. Most delayed allocation buffers are pushed out in the context of an xfsd daemon. There are a limited number of these daemons, so we can't sleep in the daemon waiting for other delayed allocation buffers to be written out. Specifically we can't wait in the xfsd daemon for the buffers covering the rest of the newly allocated

extent to be flushed, because it may be that this daemon is the only one so if we don't do it no one will.

While in performing the allocation we can log an item stating that the allocated blocks should be zeroed if we don't log anything stating otherwise, it is difficult to know when we can actually log an item stating not to perform the zeroing. We cannot log such an item until we know for sure that all of the buffers covering the allocated extent have been allocated, but we cannot wait in xfs_strat_write() for that to happen. Any solution to this problem must also address how we are going to know when everything in the allocated range has been written.

Another problem involving getting the rest of the buffers covering the allocated region flushed to disk involves the interaction with the log reservation mechanism used by all transactions. Since we are going to log a block zero item along with the allocation of the extent, the log will not be able to wrap around beyond that block zero item until we know that all of the buffers covering the allocated extent have been flushed. If getting those buffers flushed involves acquiring a log reservation then we could deadlock on the log space resource.

3.0 Solution

The solution proposed here does not live up to the goal specified in the introduction. Specifically, it requires writing zeroes to the disk even when the data for the region being zeroed resides in memory. This was not my first choice, but the restrictiveness of the environment in which the block zeroing takes place has so far left me no alternative.

The basic mechanism is described in the sections below.

3.1 Delayed Allocation Extent Conversion

A buffer over a delayed allocation extent is picked up by an xfsd to be flushed out and is passed to xfs_strat_write(). Then we take the following steps:

- Acquire the inode lock exclusively.
- While region locks overlapping our buffer exist: drop the inode lock; sleep on the region lock; reacquire the inode lock and check again.
- Once we know that we don't overlap a region, we determine the layout of the extents underlying our buffer by calling xfs_bmapi() as if we were going to read. The underlying extents can be any mix of real and delayed allocation extents.
- If there are any delayed allocation extents underlying our buffer, then create a region lock from the start of the first delayed allocation extent to the end of the last one. We can ASSERT that this does not overlap any other region locks, even if the region lock we get extends beyond the bounds of our buffer.
- Allocate a block zero item that can store the number of delayed allocation extents we are going to make real.

- For each delayed allocation extent underlying our buffer: start a transaction or inherit the transaction from11 the previous loop; add the inode to the transaction; call xfs_bmapi() to convert the delayed allocation extent to a real extent; add the region for this delayed allocation extent to the BZ item and add it to the transaction; and commit the transaction. We use a permanent log reservation transaction for this loop and always relog the inode and BZ item with each commit in order to guarantee that we don't run out of log space.
- Zero any portions of space which we allocated above but were not delayed allocation extents when we started. These will only occur because of fixed size extents and real time extents. We need to zero them before dropping the inode lock since there is not data in memory for the corresponding blocks, the blocks on disk are full of garbage, and once the inode lock is dropped the blocks become visible for reading.
- Drop the inode lock.
- Map out the region covered by our buffer and its underlying delayed allocation extents. For extents which underlie our buffer, write out the corresponding portion of the buffer. For extents which we allocated but are not encompassed by our buffer, allocate and write out a buffer of zeros. As we write the buffers chain them together so that we can wait for them later.
- Wait for all of the buffers to complete their I/O.
- Pull the BZ item from the AIL and or mark it such that it will not be added to the AIL when its transactions commit.
- Drop the region lock we acquired earlier.
- Make a transaction and log a don't block zero (DBZ) item corresponding to the BZ items logged earlier. We use the pointer to the BZ item as a key to pair up the BZ and DBZ items. We wait until after logging the DBZ item to free the BZ item so that that key cannot be reused until after the DBZ item is logged.
- Free the BZ item or mark it to be freed when its last transaction commits.

3.2 Writing out non delayed allocation buffers

A buffer NOT over a delayed allocation extent comes through the xfs_strategy() code to be written. Then we take the following steps.

- Acquire the inode lock SHARED.
- While region locks overlapping our buffer exist: drop the inode lock; sleep on the region lock; reacquire the inode lock and check again. The reason we do this is to ensure that if the underlying blocks are being zeroed we wait for the zeros to make it to disk before sending out the real data. We wouldn't want to reorder them.
- Drop the inode lock and send the buffer out to disk.

3.3 Interaction with file truncation

The truncate code will be changed to wait for any region locks in the region being freed to be dropped before freeing the underlying extents. This will ensure that we don't free blocks which are still being zeroed.

3.4 Implementation requirements

The block zeroing mechanism requires several small modifications to the current implementation. They are listed below:

- Add a block zero item and a don't block zero item to be logged to indicate blocks which may need to be zeroed in the event of a crash.
- Add a region lock mechanism to the in-core inode to ensure the proper ordering of zeros and data being written to disk. This allows us to drop the inode lock while writing the zeros.
- Modify the buffer strategy path to use and recognize the inode region locks.
- Modify the code in the file truncation path to recognize the inode region locks.
- Add functionality to the recovery code to recognize and process BZ and DBZ items
- Modify the xfs_bmapi() code to convert from delayed to real ONLY the region specified by the caller as opposed to the entire delayed allocation extent

All together these pieces incorporate a complete block zeroing solution. Each of the pieces will be discussed in more detail in the following sections.

3.4.1 Block Zero and Don't Block Zero Items

The block zero (BZ) and don't block zero (DBZ) log items will be placed in the on disk log to indicate that a portion of a file should be zeroed or no longer needs to be zeroed in the face of a system crash. The BZ item will contain an array of ranges of blocks to be zeroed or not zeroed in the on disk image. The BZ and DBZ items will both contain an inode number, inode generation number, and the pointer to the DBZ item as a key binding the two items in the log.

In memory the BZ item will adhere to the standard log item format. Its private fields will include:

- An array of offset length pairs to indicate what ranges of the file must be zeroed.
- A count of the number of entries in the array which are currently valid.
- A count of the total number of entries.

Once it is logged, a BZ item will be placed in the AIL with the lsn of its transaction. It will only move forward in the AIL when it is relogged. It will not be able to be pushed. It will only be removed from the AIL when the region it describes has been entirely flushed out or zeroed. It is important that the BZ item is removed from the AIL before the DBZ item is logged, because we may not be able to obtain a log reservation to log the DBZ item until the BZ item has been removed from the AIL.

The DBZ items will never be entered into the AIL. As soon as they have been logged to disk they are no longer needed, because it is guaranteed that the DBZ item will not be overwritten in the log until after its corresponding BZ item has been overwritten. The on disk image of the BZ item will identify its corresponding DBZ item by the using the in memory BZ item pointer as a key. We will guarantee that the BZ item is not reused until after the DBZ item has been committed.

3.4.2 Inode Region Locks

Region locks will be used to control access to portions of a file while they are being converted from delayed allocation to real extents as described above. Each lock will be described by a data structure containing:

- An offset and length in file system blocks
- A spinlock used to synchronize going to sleep on the lock
- A count of the number of procs sleeping on the lock
- Forward and backward pointers for the lock list

In addition, all region locks for a given inode will use a single semaphore pointed to by the in core inode structure. The common sleeping semaphore for all the region locks will be used to reduce the amount of memory used by the locks and to solve the problem of when the semaphore itself can be freed. Specifically, the problem is that the processes waking up on the semaphore actually reference it, so it cannot be freed until all of the sleepers have left the semaphore code. Also, the common semaphore will be dynamically allocated separately from the inode so that we only use the semaphore memory when necessary. The semaphore will be freed when the inode is eventually freed.

Since there is no way to atomically drop a multi-reader lock (the inode lock) and go to sleep on a semaphore, the following sequence will be used to go to sleep on a region lock:

- Acquire the inode lock exclusively
- Find the region lock on which we’re going to sleep, if it is not there then there is no need to sleep
- Acquire the region lock spin lock
- Bump the count of sleeping processes on the region lock
- Drop the inode lock
- Use the spunlock_psema() routine to atomically drop the spinlock and go to sleep on the inode’s semaphore

Waking up the processes sleeping on a region lock will require:

- Acquire the inode lock exclusively
- If the sleep count in the region lock is zero then there is nothing to do and the region lock can be freed
- Acquire the spin lock in the region lock to ensure that noone is in the process of going to sleep

- Remove the region lock structure from the inode's list
- Wake up all processes sleeping on the inode's semaphore

Since we wake up all the processes sleeping for any region lock whenever one is released, the processes which have awoken must check to see if they can now run. To do so they must repeat the steps described above used to go to sleep in the first place. This scheme may cause some unnecessary wakeups, but the number of region locks and processes sleeping on region locks should always be small. This is because the region locks are limited in use to the xfsds, processes trying to truncate the file, and other processes pushing out dirty buffers.

4.0 Conclusion

This paper presents a solution to the block zeroing problem for xFS. It is not a great solution performance wise, but there is plenty of room for optimization. Basically, optimizations will involve searching the chunk cache for buffers which overlap the regions we're going to write zeros to and writing the cached buffer instead. This, along with better buffer clustering algorithms will minimize the number of zeros we actually write to disk.

Coming up with a design which works within the restrictions of our transaction, locking, chunk cache, and delayed allocation mechanisms is much trickier than I originally expected. Below are listed some of the interesting "rules" encountered during the design process:

1. We really can't acquire locks before getting log reservations.
2. Any lock which we do acquire before getting a log reservation must always be acquired before getting the log reservation and must not be dropped until the transaction or sequence of chained transactions it is a part of is complete. Otherwise we end up with lock versus log space deadlocks. In other words, in a sequence of chained transactions always hold on to your locks.
3. When something is placed in the log which cannot simply be flushed out (e.g. a BZ item) we must ensure that the process which placed it there will remove it in the same sequence of instructions. Leaving it there to be processed in an asynchronous fashion is very, very dangerous. We always tend to lose our locks or the log space necessary to remove it, and then we run out of log space.
4. A daemon which is responsible for performing a certain action (e.g. flushing out buffers) cannot wait for that action on another object to complete, because it is usually the case the that daemon will be waiting for itself.
5. Transactions must be active. Waiting for an asynchronous event to occur in the middle of a transaction or sequence of chained transactions is dangerous. All of the log space tends to get used up while you're waiting.