RCU-based dcache locking model

On many workloads, the most common operation on dcache is to look up a dentry, given a parent dentry and the name of the child. Typically, for every open(), stat() etc., the dentry corresponding to the pathname will be looked up by walking the tree starting with the first component of the pathname and using that dentry along with the next component to look up the next level and so on. Since it is a frequent operation for workloads like multiuser environments and web servers, it is important to optimize this path.

Prior to 2.5.10, dcache_lock was acquired in d_lookup and thus in every component during path look-up. Since 2.5.10 onwards, fast-walk algorithm changed this by holding the dcache_lock at the beginning and walking as many cached path component dentries as possible. This significantly decreases the number of acquisition of dcache_lock. However it also increases the lock hold time significantly and affects performance in large SMP machines. Since 2.5.62 kernel, dcache has been using a new locking model that uses RCU to make dcache look-up lock-free.

The current dcache locking model is not very different from the existing dcache locking model. Prior to 2.5.62 kernel, dcache_lock protected the hash chain, d_child, d_alias, d_lru lists as well as d_inode and several other things like mount look-up. RCU-based changes affect only the way the hash chain is protected. For everything else the dcache_lock must be taken for both traversing as well as updating. The hash chain updates too take the dcache_lock. The significant change is the way d_lookup traverses the hash chain, it doesn't acquire the dcache_lock for this and rely on RCU to ensure that the dentry has not been *freed*.

Dcache locking details

For many multi-user workloads, open() and stat() on files are very frequently occurring operations. Both involve walking of path names to find the dentry corresponding to the concerned file. In 2.4 kernel, dcache_lock was held during look-up of each path component. Contention and cache-line bouncing of this global lock caused significant scalability problems. With the introduction of RCU in Linux kernel, this was worked around by making the look-up of path components during path walking lock-free.

Safe lock-free look-up of dcache hash table

Dcache is a complex data structure with the hash table entries also linked together in other lists. In 2.4 kernel, dcache_lock protected all the lists. We applied RCU only on hash chain walking. The rest of the lists are still protected by dcache_lock. Some of the important changes are :

1. The deletion from hash chain is done using hlist_del_rcu() macro which doesn't initialize next pointer of the deleted dentry and this allows us to walk safely lock-free while a deletion is happening.

2. Insertion of a dentry into the hash table is done using hlist_add_head_rcu() which take care of ordering the writes - the writes to the dentry must be visible before the dentry is inserted. This works in conjunction with hlist_for_each_rcu(), which has since been replaced by hlist_for_each_entry_rcu(), while walking the hash chain. The only requirement is that all initialization to the dentry must be done before hlist_add_head_rcu() since we don't have dcache_lock protection while traversing the hash chain. This isn't different from the existing code.

3. The dentry looked up without holding dcache_lock by cannot be returned for walking if it is unhashed. It then may have a NULL d_inode or other bogosity since RCU doesn't protect the other fields in the dentry. We therefore use a flag DCACHE_UNHASHED to indicate unhashed dentries and use this in conjunction with a per-dentry lock (d_lock). Once looked up without the dcache_lock, we acquire the per-dentry lock (d_lock) and check if the dentry is unhashed. If so, the look-up is failed. If not, the reference count of the dentry is increased and the dentry is returned.

4. Once a dentry is looked up, it must be ensured during the path walk for that component it doesn't go away. In pre-2.5.10 code, this was done holding a reference to the dentry. dcache_rcu does the same. In some sense, dcache_rcu path walking looks like the pre-2.5.10 version.

5. All dentry hash chain updates must take the dcache_lock as well as the per-dentry lock in that order. dput() does this to ensure that a dentry that has just been looked up in another CPU doesn't get deleted before dget() can be done on it.

6. There are several ways to do reference counting of RCU protected objects. One such example is in ipv4 route cache where deferred freeing (using call_rcu()) is done as soon as the reference count goes to zero. This cannot be done in the case of dentries because tearing down of dentries require blocking (dentry_iput()) which isn't supported from RCU callbacks. Instead, tearing down of dentries happen synchronously in dput(), but actual freeing happens later when RCU grace period is over. This allows safe lock-free walking of the hash chains, but a matched dentry may have been partially torn down. The checking of DCACHE_UNHASHED flag with d_lock held detects such dentries and prevents them from being returned from look-up.

Maintaining POSIX rename semantics

Since look-up of dentries is lock-free, it can race against a concurrent rename operation. For example, during rename of file A to

B, look-up of either A or B must succeed. So, if look-up of B happens after A has been removed from the hash chain but not added to the new hash chain, it may fail. Also, a comparison while the name is being written concurrently by a rename may result in false positive matches violating rename semantics. Issues related to race with rename are handled as described below :

1. Look-up can be done in two ways - d_lookup() which is safe from simultaneous renames and __d_lookup() which is not. If __d_lookup() fails, it must be followed up by a d_lookup() to correctly determine whether a dentry is in the hash table or not. d_lookup() protects look-ups using a sequence lock (rename_lock).

2. The name associated with a dentry (d_name) may be changed if a rename is allowed to happen simultaneously. To avoid memcmp() in __d_lookup() go out of bounds due to a rename and false positive comparison, the name comparison is done while holding the per-dentry lock. This prevents concurrent renames during this operation.

3. Hash table walking during look-up may move to a different bucket as the current dentry is moved to a different bucket due to rename. But we use hlists in dcache hash table and they are null-terminated. So, even if a dentry moves to a different bucket, hash chain walk will terminate. [with a list_head list, it may not since termination is when the list_head in the original bucket is reached]. Since we redo the d_parent check and compare name while holding d_lock, lock-free look-up will not race against d_move().

4. There can be a theoretical race when a dentry keeps coming back to original bucket due to double moves. Due to this look-up may consider that it has never moved and can end up in a infinite loop. But this is not any worse that theoretical livelocks we already have in the kernel.

Important guidelines for filesystem developers related to dcache_rcu

1. Existing dcache interfaces (pre-2.5.62) exported to filesystem don't change. Only dcache internal implementation changes. However filesystems *must not* delete from the dentry hash chains directly using the list macros like allowed earlier. They must use dcache APIs like d_drop() or __d_drop() depending on the situation.

2. d_flags is now protected by a per-dentry lock (d_lock). All access to d_flags must be protected by it.

3. For a hashed dentry, checking of d_count needs to be protected by d_lock.

Papers and other documentation on dcache locking

1. Scaling dcache with RCU (http://linuxjournal.com/article.php?sid=7124).

2. http://lse.sourceforge.net/locking/dcache/dcache.html