Path walking and name lookup locking

Path resolution is the finding a dentry corresponding to a path name string, by performing a path walk. Typically, for every open(), stat() etc., the path name will be resolved. Paths are resolved by walking the namespace tree, starting with the first component of the pathname (eg. root or cwd) with a known dentry, then finding the child of that dentry, which is named the next component in the path string. Then repeating the lookup from the child dentry and finding its child with the next element, and so on.

Since it is a frequent operation for workloads like multiuser environments and web servers, it is important to optimize this code.

Path walking synchronisation history:
Prior to 2.5.10, dcache_lock was acquired in d_lookup (dcache hash 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.

All the above algorithms required taking a lock and reference count on the dentry that was looked up, so that may be used as the basis for walking the next path element. This is inefficient and unscalable. It is inefficient because of the locks and atomic operations required for every dentry element slows things down. It is not scalable because many parallel applications that are path-walk intensive tend to do path lookups starting from a common dentry (usually, the root "/" or current working directory). So contention on these common path elements causes lock and cacheline queueing.

Since 2.6.38, RCU is used to make a significant part of the entire path walk (including dcache look-up) completely "store-free" (so, no locks, atomics, or even stores into cachelines of common dentries). This is known as "rcu-walk" path walking.

Path walking overview

A name string specifies a start (root directory, cwd, fd-relative) and a sequence of elements (directory entry names), which together refer to a path in the namespace. A path is represented as a (dentry, vfsmount) tuple. The name elements are sub-strings, separated by '/'.

Name lookups will want to find a particular path that a name string refers to (usually the final element, or parent of final element). This is done by taking the path given by the name's starting point (which we know in advance -- eg. current->fs->cwd or current->fs->root) as the first parent of the lookup. Then iteratively for each subsequent name element, look up the child of the current parent with the given name and if it is not the desired entry, make it the parent for the next lookup.

A parent, of course, must be a directory, and we must have appropriate permissions on the parent inode to be able to walk into it.

Turning the child into a parent for the next lookup requires more checks and procedures. Symlinks essentially substitute the symlink name for the target name in the name string, and require some recursive path walking. Mount points must be followed into (thus changing the vfsmount that subsequent path elements refer to), switching from the mount point path to the root of the particular mounted vfsmount. These behaviours are variously modified depending on the exact path walking flags.

Path walking then must, broadly, do several particular things:
- find the start point of the walk;
- perform permissions and validity checks on inodes;
- perform dcache hash name lookups on (parent, name element) tuples;
- traverse mount points;
- traverse symlinks;
- lookup and create missing parts of the path on demand.

Safe store-free look-up of dcache hash table

Dcache name lookup

In order to lookup a dcache (parent, name) tuple, we take a hash on the tuple
and use that to select a bucket in the dcache-hash table. The list of entries
in that bucket is then walked, and we do a full comparison of each entry
against our (parent, name) tuple.

The hash lists are RCU protected, so list walking is not serialised with
concurrent updates (insertion, deletion from the hash). This is a standard RCU
list application with the exception of renames, which will be covered below.

Parent and name members of a dentry, as well as its membership in the dcache
hash, and its inode are protected by the per-dentry d_lock spinlock. A
reference is taken on the dentry (while the fields are verified under d_lock),
and this stabilises its d_inode pointer and actual inode. This gives a stable
point to perform the next step of our path walk against.

These members are also protected by d_seq seqlock, although this offers
read-only protection and no durability of results, so care must be taken when
using d_seq for synchronisation (see seqcount based lookups, below).

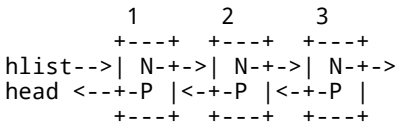
Renames

Back to the rename case. In usual RCU protected lists, the only operations that
will happen to an object is insertion, and then eventually removal from the
list. The object will not be reused until an RCU grace period is complete.
This ensures the RCU list traversal primitives can run over the object without
problems (see RCU documentation for how this works).

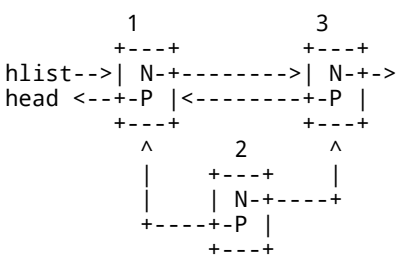
However when a dentry is renamed, its hash value can change, requiring it to be
moved to a new hash list. Allocating and inserting a new alias would be
expensive and also problematic for directory dentries. Latency would be far to
high to wait for a grace period after removing the dentry and before inserting
it in the new hash bucket. So what is done is to insert the dentry into the
new list immediately.

However, when the dentry's list pointers are updated to point to objects in the
new list before waiting for a grace period, this can result in a concurrent RCU
lookup of the old list veering off into the new (incorrect) list and missing
the remaining dentries on the list.

There is no fundamental problem with walking down the wrong list, because the
dentry comparisons will never match. However it is fatal to miss a matching
dentry. So a seqlock is used to detect when a rename has occurred, and so the
lookup can be retried.

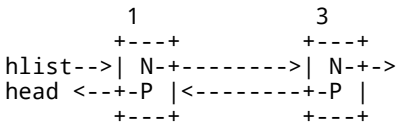


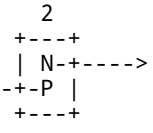
Rename of dentry 2 may require it deleted from the above list, and inserted
into a new list. Deleting 2 gives the following list.



This is a standard RCU-list deletion, which leaves the deleted object's
pointers intact, so a concurrent list walker that is currently looking at
object 2 will correctly continue to object 3 when it is time to traverse the
next object.

However, when inserting object 2 onto a new list, we end up with this:





Because we didn't wait for a grace period, there may be a concurrent lookup still at 2. Now when it follows 2's 'next' pointer, it will walk off into another list without ever having checked object 3.

A related, but distinctly different, issue is that of rename atomicity versus lookup operations. If a file is renamed from 'A' to 'B', a lookup must only find either 'A' or 'B'. So if a lookup of 'A' returns NULL, a subsequent lookup of 'B' must succeed (note the reverse is not true).

Between deleting the dentry from the old hash list, and inserting it on the new hash list, a lookup may find neither 'A' nor 'B' matching the dentry. The same rename seqlock is also used to cover this race in much the same way, by retrying a negative lookup result if a rename was in progress.

Seqcount based lookups

In refcount based dcache lookups, d_lock is used to serialise access to the dentry, stabilising it while comparing its name and parent and then taking a reference count (the reference count then gives a stable place to start the next part of the path walk from).

As explained above, we would like to do path walking without taking locks or reference counts on intermediate dentries along the path. To do this, a per dentry seqlock (d_seq) is used to take a "coherent snapshot" of what the dentry looks like (its name, parent, and inode). That snapshot is then used to start the next part of the path walk. When loading the coherent snapshot under d_seq, care must be taken to load the members up-front, and use those pointers rather than reloading from the dentry later on (otherwise we'd have interesting things like d_inode going NULL underneath us, if the name was unlinked).

Also important is to avoid performing any destructive operations (pretty much: no non-atomic stores to shared data), and to recheck the seqcount when we are "done" with the operation. Retry or abort if the seqcount does not match. Avoiding destructive or changing operations means we can easily unwind from failure.

What this means is that a caller, provided they are holding RCU lock to protect the dentry object from disappearing, can perform a seqcount based lookup which does not increment the refcount on the dentry or write to it in any way. This returned dentry can be used for subsequent operations, provided that d_seq is rechecked after that operation is complete.

Inodes are also rcu freed, so the seqcount lookup dentry's inode may also be queried for permissions.

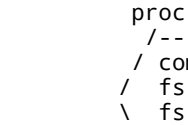
With this two parts of the puzzle, we can do path lookups without taking locks or refcounts on dentry elements.

RCU-walk path walking design
============================

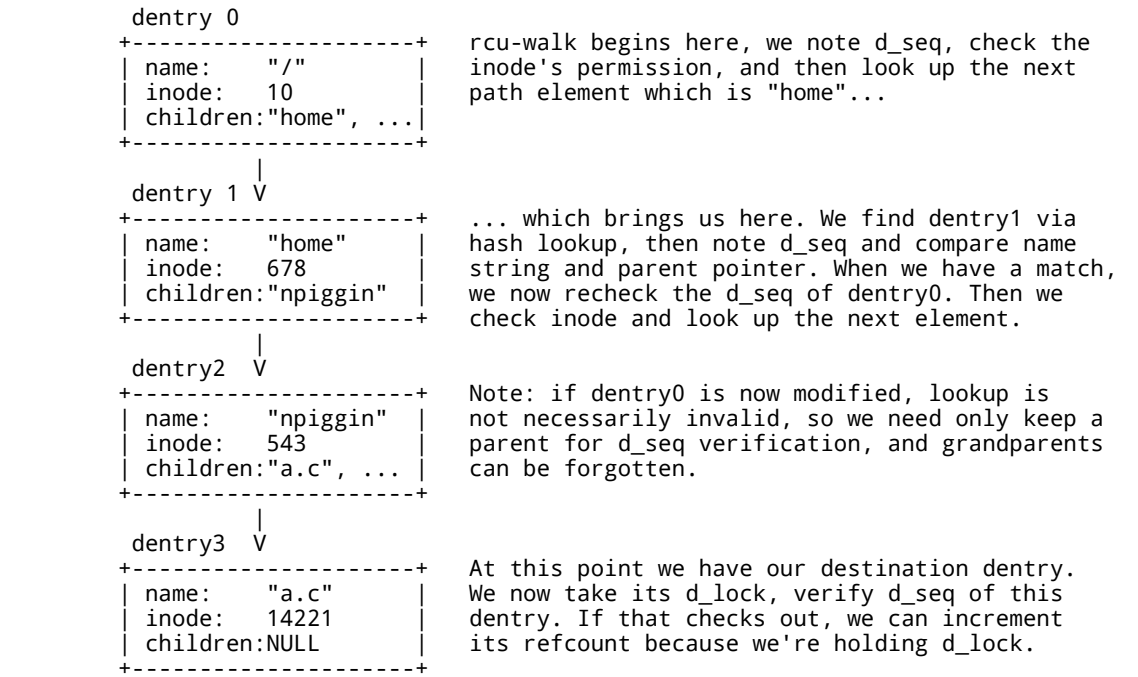
Path walking code now has two distinct modes, ref-walk and rcu-walk. ref-walk is the traditional[*] way of performing dcache lookups using d_lock to serialise concurrent modifications to the dentry and take a reference count on it. ref-walk is simple and obvious, and may sleep, take locks, etc while path walking is operating on each dentry. rcu-walk uses seqcount based dentry lookups, and can perform lookup of intermediate elements without any stores to shared data in the dentry or inode. rcu-walk can not be applied to all cases, eg. if the filesystem must sleep or perform non trivial operations, rcu-walk must be switched to ref-walk mode.

[*] RCU is still used for the dentry hash lookup in ref-walk, but not the full path walk.

Where ref-walk uses a stable, refcounted ``parent'' to walk the remaining path string, rcu-walk uses a d_seq protected snapshot. When looking up a child of this parent snapshot, we open d_seq critical section on the child before closing d_seq critical section on the parent. This gives an interlocking ladder of snapshots to walk down.



So when vi wants to open("/home/npiggin/test.c", O_RDWR), then it will
start from current->fs->root, which is a pinned dentry. Alternatively,
"./test.c" would start from cwd; both names refer to the same path in
the context of proc101.



Taking a refcount on a dentry from rcu-walk mode, by taking its d_lock,
re-checking its d_seq, and then incrementing its refcount is called
"dropping rcu" or dropping from rcu-walk into ref-walk mode.

It is, in some sense, a bit of a house of cards. If the seqcount check of the
parent snapshot fails, the house comes down, because we had closed the d_seq
section on the grandparent, so we have nothing left to stand on. In that case,
the path walk must be fully restarted (which we do in ref-walk mode, to avoid
live locks). It is costly to have a full restart, but fortunately they are
quite rare.

When we reach a point where sleeping is required, or a filesystem callout
requires ref-walk, then instead of restarting the walk, we attempt to drop rcu
at the last known good dentry we have. Avoiding a full restart in ref-walk in
these cases is fundamental for performance and scalability because blocking
operations such as creates and unlinks are not uncommon.

The detailed design for rcu-walk is like this:
* LOOKUP_RCU is set in nd->flags, which distinguishes rcu-walk from ref-walk.
* Take the RCU lock for the entire path walk, starting with the acquiring
  of the starting path (eg. root/cwd/fd-path). So now dentry refcounts are
  not required for dentry persistence.
* synchronize_rcu is called when unregistering a filesystem, so we can
  access d_ops and i_ops during rcu-walk.
* Similarly take the vfsmount lock for the entire path walk. So now mnt
  refcounts are not required for persistence. Also we are free to perform mount
  lookups, and to assume dentry mount points and mount roots are stable up and
  down the path.
* Have a per-dentry seqlock to protect the dentry name, parent, and inode,
  so we can load this tuple atomically, and also check whether any of its
  members have changed.
* Dentry lookups (based on parent, candidate string tuple) recheck the parent
  sequence after the child is found in case anything changed in the parent
  during the path walk.
* inode is also RCU protected so we can load d_inode and use the inode for
  limited things.

* i_mode, i_uid, i_gid can be tested for exec permissions during path walk.
* i_op can be loaded.
* When the destination dentry is reached, drop rcu there (ie. take d_lock,
  verify d_seq, increment refcount).
* If seqlock verification fails anywhere along the path, do a full restart
  of the path lookup in ref-walk mode. -ECHILD tends to be used (for want of
  a better errno) to signal an rcu-walk failure.

The cases where rcu-walk cannot continue are:
* NULL dentry (ie. any uncached path element)
* Following links

It may be possible eventually to make following links rcu-walk aware.

Uncached path elements will always require dropping to ref-walk mode, at the
very least because i_mutex needs to be grabbed, and objects allocated.

Final note:
"store-free" path walking is not strictly store free. We take vfsmount lock
and refcounts (both of which can be made per-cpu), and we also store to the
stack (which is essentially CPU-local), and we also have to take locks and
refcount on final dentry.

The point is that shared data, where practically possible, is not locked
or stored into. The result is massive improvements in performance and
scalability of path resolution.

Interesting statistics
======================

The following table gives rcu lookup statistics for a few simple workloads
(2s12c24t Westmere, debian non-graphical system). Ungraceful are attempts to
drop rcu that fail due to d_seq failure and requiring the entire path lookup
again. Other cases are successful rcu-drops that are required before the final
element, nodentry for missing dentry, revalidate for filesystem revalidate
routine requiring rcu drop, permission for permission check requiring drop,
and link for symlink traversal requiring drop.

	rcu-lookups	restart	nodentry	link	revalidate	permission
bootup	47121	0	4624	1010	10283	7852
dbench	25386793	0	6778659(26.7%)	55	549	1156
kbuild	2696672	10	64442(2.3%)	108764(4.0%)	1	1590
git diff	39605	0	28	2	0	106
vfstest	24185492	4945	708725(2.9%)	1076136(4.4%)	0	2651

What this shows is that failed rcu-walk lookups, ie. ones that are restarted
entirely with ref-walk, are quite rare. Even the "vfstest" case which
specifically has concurrent renames/mkdir/rmdir/ creat/unlink/etc to exercise
such races is not showing a huge amount of restarts.

Dropping from rcu-walk to ref-walk mean that we have encountered a dentry where
the reference count needs to be taken for some reason. This is either because
we have reached the target of the path walk, or because we have encountered a
condition that can't be resolved in rcu-walk mode. Ideally, we drop rcu-walk
only when we have reached the target dentry, so the other statistics show where
this does not happen.

Note that a graceful drop from rcu-walk mode due to something such as the
dentry not existing (which can be common) is not necessarily a failure of
rcu-walk scheme, because some elements of the path may have been walked in
rcu-walk mode. The further we get from common path elements (such as cwd or
root), the less contended the dentry is likely to be. The closer we are to
common path elements, the more likely they will exist in dentry cache.

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