orphan:

Making Filesystems Exportable

Overview

All filesystem operations require a dentry (or two) as a starting point. Local applications have a reference-counted hold on suitable dentries via open file descriptors or cwd/root. However remote applications that access a filesystem via a remote filesystem protocol such as NFS may not be able to hold such a reference, and so need a different way to refer to a particular dentry. As the alternative form of reference needs to be stable across renames, truncates, and server-reboot (among other things, though these tend to be the most problematic), there is no simple answer like 'filename'.

The mechanism discussed here allows each filesystem implementation to specify how to generate an opaque (outside of the filesystem) byte string for any dentry, and how to find an appropriate dentry for any given opaque byte string. This byte string will be called a "filehandle fragment" as it corresponds to part of an NFS filehandle.

A filesystem which supports the mapping between filehandle fragments and dentries will be termed "exportable".

Dcache Issues

The dcache normally contains a proper prefix of any given filesystem tree. This means that if any filesystem object is in the dcache, then all of the ancestors of that filesystem object are also in the dcache. As normal access is by filename this prefix is created naturally and maintained easily (by each object maintaining a reference count on its parent).

However when objects are included into the dcache by interpreting a filehandle fragment, there is no automatic creation of a path prefix for the object. This leads to two related but distinct features of the dcache that are not needed for normal filesystem access.

1. The dcache must sometimes contain objects that are not part of the proper prefix. i.e that are not connected to the root.
2. The dcache must be prepared for a newly found (via ->lookup) directory to already have a (non-connected) dentry, and must be able to move that dentry into place (based on the parent and name in the ->lookup). This is particularly needed for directories as it is a dcache invariant that directories only have one dentry.

To implement these features, the dcache has:

a. A dentry flag DCACHE_DISCONNECTED which is set on any dentry that might not be part of the proper prefix. This is set when anonymous dentries are created, and cleared when a dentry is noticed to be a child of a dentry which is in the proper prefix. If the refcount on a dentry with this flag set becomes zero, the dentry is immediately discarded, rather than being kept in the dcache. If a dentry that is not already in the dcache is repeatedly accessed by filehandle (as NFSD might do), an new dentry will be a allocated for each access, and discarded at the end of the access.

   Note that such a dentry can acquire children, name, ancestors, etc. without losing DCACHE_DISCONNECTED - that flag is only cleared when subtree is successfully reconnected to root. Until then dentries in such subtree are retained only as long as there are references; refcount reaching zero means immediate eviction, same as for unhashed dentries. That guarantees that we won't need to hunt them down upon umount.

b. A primitive for creation of secondary roots - d_obtain_root(inode). Those do _not_ bear DCACHE_DISCONNECTED. They are placed on the per-superblock list (->s_roots), so they can be located at umount time for eviction purposes.

c. Helper routines to allocate anonymous dentries, and to help attach loose directory dentries at lookup time. They are:

   d_obtain_alias(inode) will return a dentry for the given inode.

      If the inode already has a dentry, one of those is returned.

      If it doesn't, a new anonymous (IS_ROOT and DCACHE_DISCONNECTED) dentry is allocated and attached.

      In the case of a directory, care is taken that only one dentry can ever be attached.

   d_splice_alias(inode, dentry) will introduce a new dentry into the tree;

      either the passed-in dentry or a preexisting alias for the given inode (such as an anonymous one created by d_obtain_alias), if appropriate. It returns NULL when the passed-in dentry is used, following the calling convention of ->lookup.

Filesystem Issues

For a filesystem to be exportable it must:

1. provide the filehandle fragment routines described below.
2. make sure that d_splice_alias is used rather than d_add when ->lookup finds an inode for a given parent and name.

   If inode is NULL, d_splice_alias(inode, dentry) is equivalent to:

```
d_add(dentry, inode), NULL
```

Similarly, d_splice_alias(ERR_PTR(err), dentry) = ERR_PTR(err)

Typically the ->lookup routine will simply end with a:

```
        return d_splice_alias(inode, dentry);
}
```

A file system implementation declares that instances of the filesystem are exportable by setting the s_export_op field in the struct super_block. This field must point to a "struct export_operations" struct which has the following members:

encode_fh (optional)

> Takes a dentry and creates a filehandle fragment which can later be used to find or create a dentry for the same object. The default implementation creates a filehandle fragment that encodes a 32bit inode and generation number for the inode encoded, and if necessary the same information for the parent.

fh_to_dentry (mandatory)
: Given a filehandle fragment, this should find the implied object and create a dentry for it (possibly with d_obtain_alias).

fh_to_parent (optional but strongly recommended)
: Given a filehandle fragment, this should find the parent of the implied object and create a dentry for it (possibly with d_obtain_alias). May fail if the filehandle fragment is too small.

get_parent (optional but strongly recommended)
: When given a dentry for a directory, this should return a dentry for the parent. Quite possibly the parent dentry will have been allocated by d_alloc_anon. The default get_parent function just returns an error so any filehandle lookup that requires finding a parent will fail. ->lookup("..") is *not* used as a default as it can leave ".." entries in the dcache which are too messy to work with.

get_name (optional)
: When given a parent dentry and a child dentry, this should find a name in the directory identified by the parent dentry, which leads to the object identified by the child dentry. If no get_name function is supplied, a default implementation is provided which uses vfs_readdir to find potential names, and matches inode numbers to find the correct match.

flags
: Some filesystems may need to be handled differently than others. The export_operations struct also includes a flags field that allows the filesystem to communicate such information to nfsd. See the Export Operations Flags section below for more explanation.

A filehandle fragment consists of an array of 1 or more 4byte words, together with a one byte "type". The decode_fh routine should not depend on the stated size that is passed to it. This size may be larger than the original filehandle generated by encode_fh, in which case it will have been padded with nuls. Rather, the encode_fh routine should choose a "type" which indicates the decode_fh how much of the filehandle is valid, and how it should be interpreted.

Export Operations Flags

In addition to the operation vector pointers, struct export_operations also contains a "flags" field that allows the filesystem to communicate to nfsd that it may want to do things differently when dealing with it. The following flags are defined:

EXPORT_OP_NOWCC - disable NFSv3 WCC attributes on this filesystem
: RFC 1813 recommends that servers always send weak cache consistency (WCC) data to the client after each operation. The server should atomically collect attributes about the inode, do an operation on it, and then collect the attributes afterward. This allows the client to skip issuing GETATTRs in some situations but means that the server is calling vfs_getattr for almost all RPCs. On some filesystems (particularly those that are clustered or networked) this is expensive and atomicity is difficult to guarantee. This flag indicates to nfsd that it should skip providing WCC attributes to the client in NFSv3 replies when doing operations on this filesystem. Consider enabling this on filesystems that have an expensive ->getattr inode operation, or when atomicity between pre and post operation attribute collection is impossible to guarantee.

EXPORT_OP_NOSUBTREECHK - disallow subtree checking on this fs
: Many NFS operations deal with filehandles, which the server must then vet to ensure that they live inside of an exported tree. When the export consists of an entire filesystem, this is trivial. nfsd can just ensure that the filehandle live on the filesystem. When only part of a filesystem is exported however, then nfsd must walk the ancestors of the inode to ensure that it's within an exported subtree. This is an expensive operation and not all filesystems can support it properly. This flag exempts the filesystem from subtree checking and causes exportfs to get back an error if it tries to enable subtree checking on it.

EXPORT_OP_CLOSE_BEFORE_UNLINK - always close cached files before unlinking
: On some exportable filesystems (such as NFS) unlinking a file that is still open can cause a fair bit of extra work. For instance, the NFS client will do a "sillyrename" to ensure that the file sticks around while it's still open. When reexporting, that open file is held by nfsd so we usually end up doing a sillyrename, and then immediately deleting the sillyrenamed file just afterward when the link count actually goes to zero. Sometimes this delete can race with

other operations (for instance an rmdir of the parent directory). This flag causes nfsd to close any open files for this inode _before_ calling into the vfs to do an unlink or a rename that would replace an existing file.