IN-KERNEL CACHE OBJECT REPRESENTATION AND MANAGEMENT

By: David Howells <dhowells@redhat.com>

Contents:

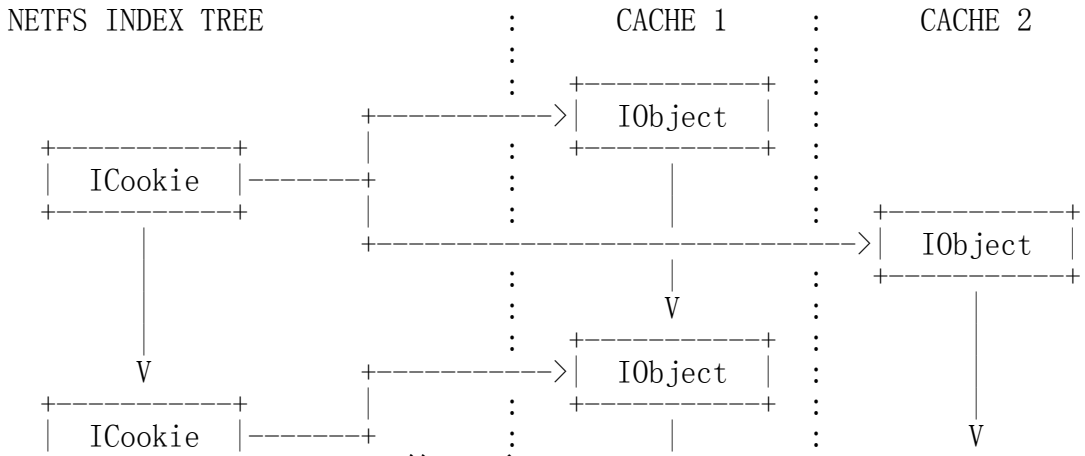
REPRESENTATION

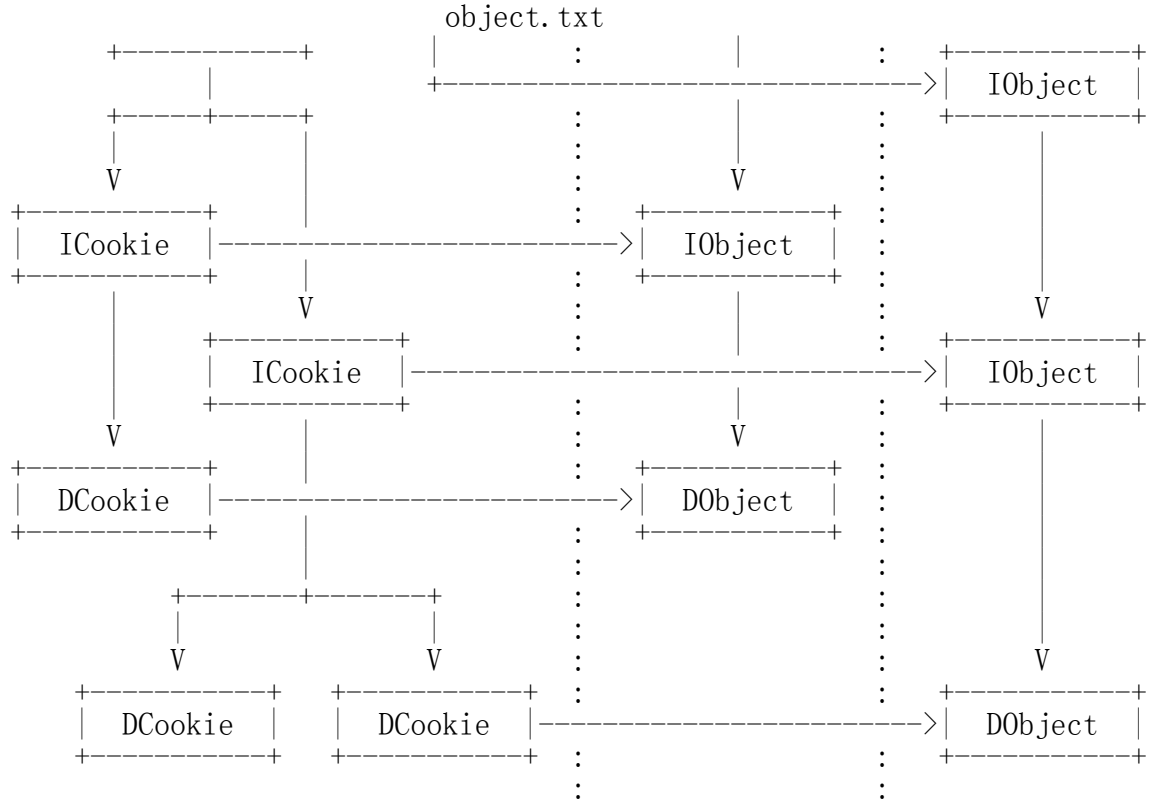
FS-Cache maintains an in-kernel representation of each object that a netfs is currently interested in. Such objects are represented by the fscache_cookie struct and are referred to as cookies.

FS-Cache also maintains a separate in-kernel representation of the objects that a cache backend is currently actively caching. Such objects are represented by the fscache_object struct. The cache backends allocate these upon request, and are expected to embed them in their own representations. These are referred to as objects.

There is a 1:N relationship between cookies and objects. A cookie may be represented by multiple objects - an index may exist in more than one cache - or even by no objects (it may not be cached).

Furthermore, both cookies and objects are hierarchical. The two hierarchies correspond, but the cookies tree is a superset of the union of the object trees of multiple caches:





In the above illustration, ICookie and IObject represent indices and DCookie and DObject represent data storage objects. Indices may have representation in multiple caches, but currently, non-index objects may not. Objects of any type may also be entirely unrepresented.

As far as the netfs API goes, the netfs is only actually permitted to see pointers to the cookies. The cookies themselves and any objects attached to those cookies are hidden from it.

==========
OBJECT MANAGEMENT STATE MACHINE
==========

Within FS-Cache, each active object is managed by its own individual state machine. The state for an object is kept in the fscache_object struct, in object->state. A cookie may point to a set of objects that are in different states.

Each state has an action associated with it that is invoked when the machine wakes up in that state. There are four logical sets of states:

(1) Preparation: states that wait for the parent objects to become ready. The representations are hierarchical, and it is expected that an object must be created or accessed with respect to its parent object.

(2) Initialisation: states that perform lookups in the cache and validate what's found and that create on disk any missing metadata.

(3) Normal running: states that allow netfs operations on objects to proceed and that update the state of objects.

(4) Termination: states that detach objects from their netfs cookies, that delete objects from disk, that handle disk and system errors and that free up in-memory resources.

In most cases, transitioning between states is in response to signalled events. When a state has finished processing, it will usually set the mask of events in which it is interested (object->event_mask) and relinquish the worker thread. Then when an event is raised (by calling fscache_raise_event()), if the event is not masked, the object will be queued for processing (by calling fscache_enqueue_object()).

PROVISION OF CPU TIME

The work to be done by the various states is given CPU time by the threads of the slow work facility (see Documentation/slow-work.txt). This is used in preference to the workqueue facility because:

(1) Threads may be completely occupied for very long periods of time by a particular work item. These state actions may be doing sequences of synchronous, journalled disk accesses (lookup, mkdir, create, setxattr, getxattr, truncate, unlink, rmdir, rename).

(2) Threads may do little actual work, but may rather spend a lot of time sleeping on I/O. This means that single-threaded and 1-per-CPU-threaded workqueues don't necessarily have the right numbers of threads.

LOCKING SIMPLIFICATION

Because only one worker thread may be operating on any particular object's state machine at once, this simplifies the locking, particularly with respect to disconnecting the netfs's representation of a cache object (fscache_cookie) from the cache backend's representation (fscache_object) - which may be requested from either end.

==========
THE SET OF STATES
==========

The object state machine has a set of states that it can be in. There are preparation states in which the object sets itself up and waits for its parent object to transit to a state that allows access to its children:

(1) State FSCACHE_OBJECT_INIT.

Initialise the object and wait for the parent object to become active. In the cache, it is expected that it will not be possible to look an object up from the parent object, until that parent object itself has been looked up.

There are initialisation states in which the object sets itself up and accesses disk for the object metadata:

(2) State FSCACHE_OBJECT_LOOKING_UP.

Look up the object on disk, using the parent as a starting point. FS-Cache expects the cache backend to probe the cache to see whether this object is represented there, and if it is, to see if it's valid (coherency management).

The cache should call fscache_object_lookup_negative() to indicate lookup failure for whatever reason, and should call fscache_obtained_object() to indicate success.

At the completion of lookup, FS-Cache will let the netfs go ahead with read operations, no matter whether the file is yet cached. If not yet cached, read operations will be immediately rejected with ENODATA until the first known page is uncached - as to that point there can be no data to be read out of the cache for that file that isn't currently also held in the pagecache.

(3) State FSCACHE_OBJECT_CREATING.

Create an object on disk, using the parent as a starting point. This happens if the lookup failed to find the object, or if the object's coherency data indicated what's on disk is out of date. In this state, FS-Cache expects the cache to create

The cache should call fscache_obtained_object() if creation completes successfully, fscache_object_lookup_negative() otherwise.

At the completion of creation, FS-Cache will start processing write operations the netfs has queued for an object. If creation failed, the write ops will be transparently discarded, and nothing recorded in the cache.

There are some normal running states in which the object spends its time servicing netfs requests:

(4) State FSCACHE_OBJECT_AVAILABLE.

A transient state in which pending operations are started, child objects are permitted to advance from FSCACHE_OBJECT_INIT state, and temporary lookup data is freed.

(5) State FSCACHE_OBJECT_ACTIVE.

The normal running state. In this state, requests the netfs makes will be passed on to the cache.

(6) State FSCACHE_OBJECT_UPDATING.

The state machine comes here to update the object in the cache from the netfs's records. This involves updating the auxiliary data that is used to maintain coherency.

And there are terminal states in which an object cleans itself up, deallocates memory and potentially deletes stuff from disk:

(7) State FSCACHE_OBJECT_LC_DYING.

The object comes here if it is dying because of a lookup or creation error. This would be due to a disk error or system error of some sort. Temporary data is cleaned up, and the parent is released.

(8) State FSCACHE_OBJECT_DYING.

The object comes here if it is dying due to an error, because its parent cookie has been relinquished by the netfs or because the cache is being withdrawn.

Any child objects waiting on this one are given CPU time so that they too can destroy themselves. This object waits for all its children to go away before advancing to the next state.

(9) State FSCACHE_OBJECT_ABORT_INIT.

The object comes to this state if it was waiting on its parent in FSCACHE_OBJECT_INIT, but its parent died. The object will destroy itself so that the parent may proceed from the FSCACHE_OBJECT_DYING state.

(10) State FSCACHE_OBJECT_RELEASING.
(11) State FSCACHE_OBJECT_RECYCLING.

The object comes to one of these two states when dying once it is rid of all its children, if it is dying because the netfs relinquished its cookie. In the first state, the cached data is expected to persist, and in the second it will be deleted.

(12) State FSCACHE_OBJECT_WITHDRAWING.

The object transits to this state if the cache decides it wants to withdraw the object from service, perhaps to make space, but also due to error or just because the whole cache is being withdrawn.

(13) State FSCACHE_OBJECT_DEAD.

The object transits to this state when the in-memory object record is ready to be deleted. The object processor shouldn't ever see an object in this state.

THE SET OF EVENTS

There are a number of events that can be raised to an object state machine:

(*) FSCACHE_OBJECT_EV_UPDATE

The netfs requested that an object be updated. The state machine will ask the cache backend to update the object, and the cache backend will ask the netfs for details of the change through its cookie definition ops.

(*) FSCACHE_OBJECT_EV_CLEARED

This is signalled in two circumstances:

(a) when an object's last child object is dropped and

(b) when the last operation outstanding on an object is completed.

This is used to proceed from the dying state.

(*) FSCACHE_OBJECT_EV_ERROR

This is signalled when an I/O error occurs during the processing of some object.

(*) FSCACHE_OBJECT_EV_RELEASE
(*) FSCACHE_OBJECT_EV_RETIRE

These are signalled when the netfs relinquishes a cookie it was using. The event selected depends on whether the netfs asks for the backing object to be retired (deleted) or retained.

(*) FSCACHE_OBJECT_EV_WITHDRAW

This is signalled when the cache backend wants to withdraw an object. This means that the object will have to be detached from the netfs's cookie.

Because the withdrawing releasing/retiring events are all handled by the object state machine, it doesn't matter if there's a collision with both ends trying to sever the connection at the same time. The state machine can just pick which one it wants to honour, and that effects the other.