Network Filesystem Caching API

Fscache provides an API by which a network filesystem can make use of local caching facilities. The API is arranged around a number of principles:

1. A cache is logically organised into volumes and data storage objects within those volumes.
2. Volumes and data storage objects are represented by various types of cookie.
3. Cookies have keys that distinguish them from their peers.
4. Cookies have coherency data that allows a cache to determine if the cached data is still valid.
5. I/O is done asynchronously where possible.

This API is used by:

```
#include <linux/fscache.h>.
```

Overview

The fscache hierarchy is organised on two levels from a network filesystem's point of view. The upper level represents "volumes" and the lower level represents "data storage objects". These are represented by two types of cookie, hereafter referred to as "volume cookies" and "cookies".

A network filesystem acquires a volume cookie for a volume using a volume key, which represents all the information that defines that volume (e.g. cell name or server address, volume ID or share name). This must be rendered as a printable string that can be used as a directory name (ie. no '/' characters and shouldn't begin with a '.'). The maximum name length is one less than the maximum size of a filename component (allowing the cache backend one char for its own purposes).

A filesystem would typically have a volume cookie for each superblock.

The filesystem then acquires a cookie for each file within that volume using an object key. Object keys are binary blobs and only need to be unique within their parent volume. The cache backend is reponsible for rendering the binary blob into something it can use and may employ hash tables, trees or whatever to improve its ability to find an object. This is transparent to the network filesystem.

A filesystem would typically have a cookie for each inode, and would acquire it in iget and relinquish it when evicting the cookie.

Once it has a cookie, the filesystem needs to mark the cookie as being in use. This causes fscache to send the cache backend off to look up/create resources for the cookie in the background, to check its coherency and, if necessary, to mark the object as being under modification.

A filesystem would typically "use" the cookie in its file open routine and unuse it in file release and it needs to use the cookie around calls to truncate the cookie locally. It *also* needs to use the cookie when the pagecache becomes dirty and unuse it when writeback is complete. This is slightly tricky, and provision is made for it.

When performing a read, write or resize on a cookie, the filesystem must first begin an operation. This copies the resources into a holding struct and puts extra pins into the cache to stop cache withdrawal from tearing down the structures being used. The actual operation can then be issued and conflicting invalidations can be detected upon completion.

The filesystem is expected to use netfslib to access the cache, but that's not actually required and it can use the fscache I/O API directly.

Volume Registration

The first step for a network filsystem is to acquire a volume cookie for the volume it wants to access:

```
struct fscache_volume *
fscache_acquire_volume(const char *volume_key,
                       const char *cache_name,
                       const void *coherency_data,
                       size_t coherency_len);
```

This function creates a volume cookie with the specified volume key as its name and notes the coherency data.

The volume key must be a printable string with no '/' characters in it. It should begin with the name of the filesystem and should be no longer than 254 characters. It should uniquely represent the volume and will be matched with what's stored in the cache.

The caller may also specify the name of the cache to use. If specified, fscache will look up or create a cache cookie of that name and will use a cache of that name if it is online or comes online. If no cache name is specified, it will use the first cache that comes to hand and set the name to that.

The specified coherency data is stored in the cookie and will be matched against coherency data stored on disk. The data pointer may be NULL if no data is provided. If the coherency data doesn't match, the entire cache volume will be invalidated.

This function can return errors such as EBUSY if the volume key is already in use by an acquired volume or ENOMEM if an allocation failure occured. It may also return a NULL volume cookie if fscache is not enabled. It is safe to pass a NULL cookie to

any function that takes a volume cookie. This will cause that function to do nothing.

When the network filesystem has finished with a volume, it should relinquish it by calling:

```
void fscache_relinquish_volume(struct fscache_volume *volume,
                               const void *coherency_data,
                               bool invalidate);
```

This will cause the volume to be committed or removed, and if sealed the coherency data will be set to the value supplied. The amount of coherency data must match the length specified when the volume was acquired. Note that all data cookies obtained in this volume must be relinquished before the volume is relinquished.

Data File Registration

Once it has a volume cookie, a network filesystem can use it to acquire a cookie for data storage:

```
struct fscache_cookie *
fscache_acquire_cookie(struct fscache_volume *volume,
                       u8 advice,
                       const void *index_key,
                       size_t index_key_len,
                       const void *aux_data,
                       size_t aux_data_len,
                       loff_t object_size)
```

This creates the cookie in the volume using the specified index key. The index key is a binary blob of the given length and must be unique for the volume. This is saved into the cookie. There are no restrictions on the content, but its length shouldn't exceed about three quarters of the maximum filename length to allow for encoding.

The caller should also pass in a piece of coherency data in aux_data. A buffer of size aux_data_len will be allocated and the coherency data copied in. It is assumed that the size is invariant over time. The coherency data is used to check the validity of data in the cache. Functions are provided by which the coherency data can be updated.

The file size of the object being cached should also be provided. This may be used to trim the data and will be stored with the coherency data.

This function never returns an error, though it may return a NULL cookie on allocation failure or if fscache is not enabled. It is safe to pass in a NULL volume cookie and pass the NULL cookie returned to any function that takes it. This will cause that function to do nothing.

When the network filesystem has finished with a cookie, it should relinquish it by calling:

```
void fscache_relinquish_cookie(struct fscache_cookie *cookie,
                               bool retire);
```

This will cause fscache to either commit the storage backing the cookie or delete it.

Marking A Cookie In-Use

Once a cookie has been acquired by a network filesystem, the filesystem should tell fscache when it intends to use the cookie (typically done on file open) and should say when it has finished with it (typically on file close):

```
void fscache_use_cookie(struct fscache_cookie *cookie,
                        bool will_modify);
void fscache_unuse_cookie(struct fscache_cookie *cookie,
                          const void *aux_data,
                          const loff_t *object_size);
```

The *use* function tells fscache that it will use the cookie and, additionally, indicate if the user is intending to modify the contents locally. If not yet done, this will trigger the cache backend to go and gather the resources it needs to access/store data in the cache. This is done in the background, and so may not be complete by the time the function returns.

The *unuse* function indicates that a filesystem has finished using a cookie. It optionally updates the stored coherency data and object size and then decreases the in-use counter. When the last user unuses the cookie, it is scheduled for garbage collection. If not reused within a short time, the resources will be released to reduce system resource consumption.

A cookie must be marked in-use before it can be accessed for read, write or resize - and an in-use mark must be kept whilst there is dirty data in the pagecache in order to avoid an oops due to trying to open a file during process exit.

Note that in-use marks are cumulative. For each time a cookie is marked in-use, it must be unused.

Resizing A Data File (Truncation)

If a network filesystem file is resized locally by truncation, the following should be called to notify the cache:

```
void fscache_resize_cookie(struct fscache_cookie *cookie,
                           loff_t new_size);
```

The caller must have first marked the cookie in-use. The cookie and the new size are passed in and the cache is synchronously resized. This is expected to be called from `->setattr()` inode operation under the inode lock.

Data I/O API

To do data I/O operations directly through a cookie, the following functions are available:

```
int fscache_begin_read_operation(struct netfs_cache_resources *cres,
                                 struct fscache_cookie *cookie);
int fscache_read(struct netfs_cache_resources *cres,
                 loff_t start_pos,
                 struct iov_iter *iter,
                 enum netfs_read_from_hole read_hole,
                 netfs_io_terminated_t term_func,
                 void *term_func_priv);
int fscache_write(struct netfs_cache_resources *cres,
                  loff_t start_pos,
                  struct iov_iter *iter,
                  netfs_io_terminated_t term_func,
                  void *term_func_priv);
```

The *begin* function sets up an operation, attaching the resources required to the cache resources block from the cookie. Assuming it doesn't return an error (for instance, it will return -ENOBUFS if given a NULL cookie, but otherwise do nothing), then one of the other two functions can be issued.

The *read* and *write* functions initiate a direct-IO operation. Both take the previously set up cache resources block, an indication of the start file position, and an I/O iterator that describes buffer and indicates the amount of data.

The read function also takes a parameter to indicate how it should handle a partially populated region (a hole) in the disk content. This may be to ignore it, skip over an initial hole and place zeros in the buffer or give an error.

The read and write functions can be given an optional termination function that will be run on completion:

```
typedef
void (*netfs_io_terminated_t)(void *priv, ssize_t transferred_or_error,
                              bool was_async);
```

If a termination function is given, the operation will be run asynchronously and the termination function will be called upon completion. If not given, the operation will be run synchronously. Note that in the asynchronous case, it is possible for the operation to complete before the function returns.

Both the read and write functions end the operation when they complete, detaching any pinned resources.

The read operation will fail with ESTALE if invalidation occurred whilst the operation was ongoing.

Data File Coherency

To request an update of the coherency data and file size on a cookie, the following should be called:

```
void fscache_update_cookie(struct fscache_cookie *cookie,
                           const void *aux_data,
                           const loff_t *object_size);
```

This will update the cookie's coherency data and/or file size.

Data File Invalidation

Sometimes it will be necessary to invalidate an object that contains data. Typically this will be necessary when the server informs the network filesystem of a remote third-party change - at which point the filesystem has to throw away the state and cached data that it had for an file and reload from the server.

To indicate that a cache object should be invalidated, the following should be called:

```
void fscache_invalidate(struct fscache_cookie *cookie,
                        const void *aux_data,
                        loff_t size,
                        unsigned int flags);
```

This increases the invalidation counter in the cookie to cause outstanding reads to fail with -ESTALE, sets the coherency data and file size from the information supplied, blocks new I/O on the cookie and dispatches the cache to go and get rid of the old data.

Invalidation runs asynchronously in a worker thread so that it doesn't block too much.

Write-Back Resource Management

To write data to the cache from network filesystem writeback, the cache resources required need to be pinned at the point the

modification is made (for instance when the page is marked dirty) as it's not possible to open a file in a thread that's exiting.

The following facilities are provided to manage this:

- An inode flag, `I_PINNING_FSCACHE_WB`, is provided to indicate that an in-use is held on the cookie for this inode. It can only be changed if the the inode lock is held.
- A flag, `unpinned_fscache_wb` is placed in the `writeback_control` struct that gets set if `__writeback_single_inode()` clears `I_PINNING_FSCACHE_WB` because all the dirty pages were cleared.

To support this, the following functions are provided:

```
bool fscache_dirty_folio(struct address_space *mapping,
                         struct folio *folio,
                         struct fscache_cookie *cookie);
void fscache_unpin_writeback(struct writeback_control *wbc,
                             struct fscache_cookie *cookie);
void fscache_clear_inode_writeback(struct fscache_cookie *cookie,
                                   struct inode *inode,
                                   const void *aux);
```

The *set* function is intended to be called from the filesystem's `dirty_folio` address space operation. If `I_PINNING_FSCACHE_WB` is not set, it sets that flag and increments the use count on the cookie (the caller must already have called `fscache_use_cookie()`).

The *unpin* function is intended to be called from the filesystem's `write_inode` superblock operation. It cleans up after writing by unusing the cookie if unpinned_fscache_wb is set in the writeback_control struct.

The *clear* function is intended to be called from the netfs's `evict_inode` superblock operation. It must be called *after* `truncate_inode_pages_final()`, but *before* `clear_inode()`. This cleans up any hanging `I_PINNING_FSCACHE_WB`. It also allows the coherency data to be updated.

Caching of Local Modifications

If a network filesystem has locally modified data that it wants to write to the cache, it needs to mark the pages to indicate that a write is in progress, and if the mark is already present, it needs to wait for it to be removed first (presumably due to an already in-progress operation). This prevents multiple competing DIO writes to the same storage in the cache.

Firstly, the netfs should determine if caching is available by doing something like:

```
bool caching = fscache_cookie_enabled(cookie);
```

If caching is to be attempted, pages should be waited for and then marked using the following functions provided by the netfs helper library:

```
void set_page_fscache(struct page *page);
void wait_on_page_fscache(struct page *page);
int wait_on_page_fscache_killable(struct page *page);
```

Once all the pages in the span are marked, the netfs can ask fscache to schedule a write of that region:

```
void fscache_write_to_cache(struct fscache_cookie *cookie,
                            struct address_space *mapping,
                            loff_t start, size_t len, loff_t i_size,
                            netfs_io_terminated_t term_func,
                            void *term_func_priv,
                            bool caching)
```

And if an error occurs before that point is reached, the marks can be removed by calling:

```
void fscache_clear_page_bits(struct fscache_cookie *cookie,
                             struct address_space *mapping,
                             loff_t start, size_t len,
                             bool caching)
```

In both of these functions, the cookie representing the cache object to be written to and a pointer to the mapping to which the source pages are attached are passed in; start and len indicate the size of the region that's going to be written (it doesn't have to align to page boundaries necessarily, but it does have to align to DIO boundaries on the backing filesystem). The caching parameter indicates if caching should be skipped, and if false, the functions do nothing.

The write function takes some additional parameters: i_size indicates the size of the netfs file and term_func indicates an optional completion function, to which term_func_priv will be passed, along with the error or amount written.

Note that the write function will always run asynchronously and will unmark all the pages upon completion before calling term_func.

Page Release and Invalidation

Fscache keeps track of whether we have any data in the cache yet for a cache object we've just created. It knows it doesn't have to do any reading until it has done a write and then the page it wrote from has been released by the VM, after which it *has* to look in the

cache.

To inform fscache that a page might now be in the cache, the following function should be called from the `releasepage` address space op:

```
void fscache_note_page_release(struct fscache_cookie *cookie);
```

if the page has been released (ie. releasepage returned true).

Page release and page invalidation should also wait for any mark left on the page to say that a DIO write is underway from that page:

```
void wait_on_page_fscache(struct page *page);
int wait_on_page_fscache_killable(struct page *page);
```

API Function Reference

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\caching\(linux-master)(Documentation)(filesystems)(caching)netfs-api.rst, line 453)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/fscache.h
```