FS-CACHE CACHE BACKEND API

The FS-Cache system provides an API by which actual caches can be supplied to FS-Cache for it to then serve out to network filesystems and other interested parties.

This API is declared in <linux/fscache-cache.h>.

INITIALISING AND REGISTERING A CACHE

To start off, a cache definition must be initialised and registered for each cache the backend wants to make available. For instance, CacheFS does this in the fill_super() operation on mounting.

The cache definition (struct fscache_cache) should be initialised by calling:

```
	void fscache_init_cache(struct fscache_cache *cache,
				struct fscache_cache_ops *ops,
				const char *idfmt,
				...);
```

Where:

 (*) "cache" is a pointer to the cache definition;

 (*) "ops" is a pointer to the table of operations that the backend supports on this cache; and

 (*) "idfmt" is a format and printf-style arguments for constructing a label for the cache.

The cache should then be registered with FS-Cache by passing a pointer to the previously initialised cache definition to:

```
	int fscache_add_cache(struct fscache_cache *cache,
			      struct fscache_object *fsdef,
			      const char *tagname);
```

Two extra arguments should also be supplied:

 (*) "fsdef" which should point to the object representation for the FS-Cache master index in this cache. Netfs primary index entries will be created here. FS-Cache keeps the caller's reference to the index object if successful and will release it upon withdrawal of the cache.

 (*) "tagname" which, if given, should be a text string naming this cache. If this is NULL, the identifier will be used instead. For CacheFS, the identifier is set to name the underlying block device and the tag can be supplied by mount.

This function may return -ENOMEM if it ran out of memory or -EEXIST if the tag is already in use. 0 will be returned on success.

```
=====================
UNREGISTERING A CACHE
=====================
```

A cache can be withdrawn from the system by calling this function with a pointer to the cache definition:

```
	void fscache_withdraw_cache(struct fscache_cache *cache);
```

In CacheFS's case, this is called by put_super().

```
========
SECURITY
========
```

The cache methods are executed one of two contexts:

 (1) that of the userspace process that issued the netfs operation that caused
     the cache method to be invoked, or

 (2) that of one of the processes in the FS-Cache thread pool.

In either case, this may not be an appropriate context in which to access the cache.

The calling process's fsuid, fsgid and SELinux security identities may need to be masqueraded for the duration of the cache driver's access to the cache.
This is left to the cache to handle; FS-Cache makes no effort in this regard.

```
===================================
CONTROL AND STATISTICS PRESENTATION
===================================
```

The cache may present data to the outside world through FS-Cache's interfaces in sysfs and procfs - the former for control and the latter for statistics.

A sysfs directory called /sys/fs/fscache/<cachetag>/ is created if CONFIG_SYSFS is enabled. This is accessible through the kobject struct fscache_cache::kobj and is for use by the cache as it sees fit.

```
========================
RELEVANT DATA STRUCTURES
========================
```

 (*) Index/Data file FS-Cache representation cookie:

```
	struct fscache_cookie {
		struct fscache_object_def	*def;
		struct fscache_netfs		*netfs;
```

```
                void                            *netfs_data;
                ...
        };
```

The fields that might be of use to the backend describe the object definition, the netfs definition and the netfs's data for this cookie. The object definition contain functions supplied by the netfs for loading and matching index entries; these are required to provide some of the cache operations.

(*) In-cache object representation:

```
        struct fscache_object {
                int                             debug_id;
                enum {
                        FSCACHE_OBJECT_RECYCLING,
                        ...
                }                               state;
                spinlock_t                      lock
                struct fscache_cache            *cache;
                struct fscache_cookie           *cookie;
                ...
        };
```

Structures of this type should be allocated by the cache backend and passed to FS-Cache when requested by the appropriate cache operation. In the case of CacheFS, they're embedded in CacheFS's internal object structures.

The debug_id is a simple integer that can be used in debugging messages that refer to a particular object. In such a case it should be printed using "OBJ%x" to be consistent with FS-Cache.

Each object contains a pointer to the cookie that represents the object it is backing. An object should retired when put_object() is called if it is in state FSCACHE_OBJECT_RECYCLING. The fscache_object struct should be initialised by calling fscache_object_init(object).

(*) FS-Cache operation record:

```
        struct fscache_operation {
                atomic_t                usage;
                struct fscache_object   *object;
                unsigned long           flags;
        #define FSCACHE_OP_EXCLUSIVE
                void (*processor)(struct fscache_operation *op);
                void (*release)(struct fscache_operation *op);
                ...
        };
```

FS-Cache has a pool of threads that it uses to give CPU time to the various asynchronous operations that need to be done as part of driving the cache. These are represented by the above structure. The processor method is called to give the op CPU time, and the release method to get rid of it when its usage count reaches 0.

An operation can be made exclusive upon an object by setting the appropriate flag before enqueuing it with fscache_enqueue_operation(). If an operation needs more processing time, it should be enqueued again.

(*) FS-Cache retrieval operation record:

```
	struct fscache_retrieval {
		struct fscache_operation op;
		struct address_space	*mapping;
		struct list_head	*to_do;
		...
	};
```

A structure of this type is allocated by FS-Cache to record retrieval and allocation requests made by the netfs. This struct is then passed to the backend to do the operation. The backend may get extra refs to it by calling fscache_get_retrieval() and refs may be discarded by calling fscache_put_retrieval().

A retrieval operation can be used by the backend to do retrieval work. To do this, the retrieval->op.processor method pointer should be set appropriately by the backend and fscache_enqueue_retrieval() called to submit it to the thread pool. CacheFiles, for example, uses this to queue page examination when it detects PG_lock being cleared.

The to_do field is an empty list available for the cache backend to use as it sees fit.

(*) FS-Cache storage operation record:

```
	struct fscache_storage {
		struct fscache_operation op;
		pgoff_t			store_limit;
		...
	};
```

A structure of this type is allocated by FS-Cache to record outstanding writes to be made. FS-Cache itself enqueues this operation and invokes the write_page() method on the object at appropriate times to effect storage.

================
CACHE OPERATIONS
================

The cache backend provides FS-Cache with a table of operations that can be performed on the denizens of the cache. These are held in a structure of type:

```
	struct fscache_cache_ops
```

(*) Name of cache provider [mandatory]:

```
   const char *name
```

This isn't strictly an operation, but should be pointed at a string naming the backend.

(*) Allocate a new object [mandatory]:

```
   struct fscache_object *(*alloc_object)(struct fscache_cache *cache,
                                          struct fscache_cookie *cookie)
```

This method is used to allocate a cache object representation to back a cookie in a particular cache. fscache_object_init() should be called on the object to initialise it prior to returning.

This function may also be used to parse the index key to be used for multiple lookup calls to turn it into a more convenient form. FS-Cache will call the lookup_complete() method to allow the cache to release the form once lookup is complete or aborted.

(*) Look up and create object [mandatory]:

```
   void (*lookup_object)(struct fscache_object *object)
```

This method is used to look up an object, given that the object is already allocated and attached to the cookie. This should instantiate that object in the cache if it can.

The method should call fscache_object_lookup_negative() as soon as possible if it determines the object doesn't exist in the cache. If the object is found to exist and the netfs indicates that it is valid then fscache_obtained_object() should be called once the object is in a position to have data stored in it. Similarly, fscache_obtained_object() should also be called once a non-present object has been created.

If a lookup error occurs, fscache_object_lookup_error() should be called to abort the lookup of that object.

(*) Release lookup data [mandatory]:

```
   void (*lookup_complete)(struct fscache_object *object)
```

This method is called to ask the cache to release any resources it was using to perform a lookup.

(*) Increment object refcount [mandatory]:

```
   struct fscache_object *(*grab_object)(struct fscache_object *object)
```

This method is called to increment the reference count on an object. It may fail (for instance if the cache is being withdrawn) by returning NULL. It should return the object pointer if successful.

(*) Lock/Unlock object [mandatory]:

```
        void (*lock_object)(struct fscache_object *object)
        void (*unlock_object)(struct fscache_object *object)
```

These methods are used to exclusively lock an object. It must be possible to schedule with the lock held, so a spinlock isn't sufficient.

(*) Pin/Unpin object [optional]:

```
        int (*pin_object)(struct fscache_object *object)
        void (*unpin_object)(struct fscache_object *object)
```

These methods are used to pin an object into the cache. Once pinned an object cannot be reclaimed to make space. Return -ENOSPC if there's not enough space in the cache to permit this.

(*) Update object [mandatory]:

```
        int (*update_object)(struct fscache_object *object)
```

This is called to update the index entry for the specified object. The new information should be in object->cookie->netfs_data. This can be obtained by calling object->cookie->def->get_aux()/get_attr().

(*) Discard object [mandatory]:

```
        void (*drop_object)(struct fscache_object *object)
```

This method is called to indicate that an object has been unbound from its cookie, and that the cache should release the object's resources and retire it if it's in state FSCACHE_OBJECT_RECYCLING.

This method should not attempt to release any references held by the caller. The caller will invoke the put_object() method as appropriate.

(*) Release object reference [mandatory]:

```
        void (*put_object)(struct fscache_object *object)
```

This method is used to discard a reference to an object. The object may be freed when all the references to it are released.

(*) Synchronise a cache [mandatory]:

```
        void (*sync)(struct fscache_cache *cache)
```

This is called to ask the backend to synchronise a cache with its backing device.

(*) Dissociate a cache [mandatory]:

```
	void (*dissociate_pages)(struct fscache_cache *cache)
```

This is called to ask a cache to perform any page dissociations as part of cache withdrawal.

(*) Notification that the attributes on a netfs file changed [mandatory]:

```
	int (*attr_changed)(struct fscache_object *object);
```

This is called to indicate to the cache that certain attributes on a netfs file have changed (for example the maximum size a file may reach). The cache can read these from the netfs by calling the cookie's get_attr() method.

The cache may use the file size information to reserve space on the cache. It should also call fscache_set_store_limit() to indicate to FS-Cache the highest byte it's willing to store for an object.

This method may return -ve if an error occurred or the cache object cannot be expanded. In such a case, the object will be withdrawn from service.

This operation is run asynchronously from FS-Cache's thread pool, and storage and retrieval operations from the netfs are excluded during the execution of this operation.

(*) Reserve cache space for an object's data [optional]:

```
	int (*reserve_space)(struct fscache_object *object, loff_t size);
```

This is called to request that cache space be reserved to hold the data for an object and the metadata used to track it. Zero size should be taken as request to cancel a reservation.

This should return 0 if successful, -ENOSPC if there isn't enough space available, or -ENOMEM or -EIO on other errors.

The reservation may exceed the current size of the object, thus permitting future expansion. If the amount of space consumed by an object would exceed the reservation, it's permitted to refuse requests to allocate pages, but not required. An object may be pruned down to its reservation size if larger than that already.

(*) Request page be read from cache [mandatory]:

```
	int (*read_or_alloc_page)(struct fscache_retrieval *op,
				  struct page *page,
				  gfp_t gfp)
```

This is called to attempt to read a netfs page from the cache, or to

```
	reserve a backing block if not.  FS-Cache will have done as much checking
	as it can before calling, but most of the work belongs to the backend.

	If there's no page in the cache, then -ENODATA should be returned if the
	backend managed to reserve a backing block; -ENOBUFS or -ENOMEM if it
	didn't.

	If there is suitable data in the cache, then a read operation should be
	queued and 0 returned.  When the read finishes, fscache_end_io() should be
	called.

	The fscache_mark_pages_cached() should be called for the page if any cache
	metadata is retained.  This will indicate to the netfs that the page needs
	explicit uncaching.  This operation takes a pagevec, thus allowing several
	pages to be marked at once.

	The retrieval record pointed to by op should be retained for each page
	queued and released when I/O on the page has been formally ended.
	fscache_get/put_retrieval() are available for this purpose.

	The retrieval record may be used to get CPU time via the FS-Cache thread
	pool.  If this is desired, the op->op.processor should be set to point to
	the appropriate processing routine, and fscache_enqueue_retrieval() should
	be called at an appropriate point to request CPU time.  For instance, the
	retrieval routine could be enqueued upon the completion of a disk read.
	The to_do field in the retrieval record is provided to aid in this.

	If an I/O error occurs, fscache_io_error() should be called and -ENOBUFS
	returned if possible or fscache_end_io() called with a suitable error
	code..


 (*) Request pages be read from cache [mandatory]:

	int (*read_or_alloc_pages)(struct fscache_retrieval *op,
				   struct list_head *pages,
				   unsigned *nr_pages,
				   gfp_t gfp)

	This is like the read_or_alloc_page() method, except it is handed a list
	of pages instead of one page.  Any pages on which a read operation is
	started must be added to the page cache for the specified mapping and also
	to the LRU.  Such pages must also be removed from the pages list and
	*nr_pages decremented per page.

	If there was an error such as -ENOMEM, then that should be returned; else
	if one or more pages couldn't be read or allocated, then -ENOBUFS should
	be returned; else if one or more pages couldn't be read, then -ENODATA
	should be returned.  If all the pages are dispatched then 0 should be
	returned.


 (*) Request page be allocated in the cache [mandatory]:

	int (*allocate_page)(struct fscache_retrieval *op,
			     struct page *page,
                              gfp_t gfp)
```

This is like the read_or_alloc_page() method, except that it shouldn't
read from the cache, even if there's data there that could be retrieved.
It should, however, set up any internal metadata required such that
the write_page() method can write to the cache.

If there's no backing block available, then -ENOBUFS should be returned
(or -ENOMEM if there were other problems). If a block is successfully
allocated, then the netfs page should be marked and 0 returned.

(*) Request pages be allocated in the cache [mandatory]:

```
	int (*allocate_pages)(struct fscache_retrieval *op,
			      struct list_head *pages,
			      unsigned *nr_pages,
			      gfp_t gfp)
```

This is an multiple page version of the allocate_page() method. pages and
nr_pages should be treated as for the read_or_alloc_pages() method.

(*) Request page be written to cache [mandatory]:

```
	int (*write_page)(struct fscache_storage *op,
			  struct page *page);
```

This is called to write from a page on which there was a previously
successful read_or_alloc_page() call or similar. FS-Cache filters out
pages that don't have mappings.

This method is called asynchronously from the FS-Cache thread pool. It is
not required to actually store anything, provided -ENODATA is then
returned to the next read of this page.

If an error occurred, then a negative error code should be returned,
otherwise zero should be returned. FS-Cache will take appropriate action
in response to an error, such as withdrawing this object.

If this method returns success then FS-Cache will inform the netfs
appropriately.

(*) Discard retained per-page metadata [mandatory]:

```
	void (*uncache_page)(struct fscache_object *object, struct page *page)
```

This is called when a netfs page is being evicted from the pagecache. The
cache backend should tear down any internal representation or tracking it
maintains for this page.

==================
FS-CACHE UTILITIES
==================

```
FS-Cache provides some utilities that a cache backend may make use of:

 (*) Note occurrence of an I/O error in a cache:

	void fscache_io_error(struct fscache_cache *cache)

     This tells FS-Cache that an I/O error occurred in the cache.  After this
     has been called, only resource dissociation operations (object and page
     release) will be passed from the netfs to the cache backend for the
     specified cache.

     This does not actually withdraw the cache.  That must be done separately.


 (*) Invoke the retrieval I/O completion function:

	void fscache_end_io(struct fscache_retrieval *op, struct page *page,
			    int error);

     This is called to note the end of an attempt to retrieve a page.  The
     error value should be 0 if successful and an error otherwise.


 (*) Set highest store limit:

	void fscache_set_store_limit(struct fscache_object *object,
				     loff_t i_size);

     This sets the limit FS-Cache imposes on the highest byte it's willing to
     try and store for a netfs.  Any page over this limit is automatically
     rejected by fscache_read_alloc_page() and co with -ENOBUFS.


 (*) Mark pages as being cached:

	void fscache_mark_pages_cached(struct fscache_retrieval *op,
				       struct pagevec *pagevec);

     This marks a set of pages as being cached.  After this has been called,
     the netfs must call fscache_uncache_page() to unmark the pages.


 (*) Perform coherency check on an object:

	enum fscache_checkaux fscache_check_aux(struct fscache_object *object,
						const void *data,
						uint16_t datalen);

     This asks the netfs to perform a coherency check on an object that has
     just been looked up.  The cookie attached to the object will determine the
     netfs to use.  data and datalen should specify where the auxiliary data
     retrieved from the cache can be found.

     One of three values will be returned:
```

```
	(*) FSCACHE_CHECKAUX_OKAY

	    The coherency data indicates the object is valid as is.

	(*) FSCACHE_CHECKAUX_NEEDS_UPDATE

	    The coherency data needs updating, but otherwise the object is
	    valid.

	(*) FSCACHE_CHECKAUX_OBSOLETE

	    The coherency data indicates that the object is obsolete and should
	    be discarded.


 (*) Initialise a freshly allocated object:

	void fscache_object_init(struct fscache_object *object);

     This initialises all the fields in an object representation.


 (*) Indicate the destruction of an object:

	void fscache_object_destroyed(struct fscache_cache *cache);

     This must be called to inform FS-Cache that an object that belonged to a
     cache has been destroyed and deallocated.  This will allow continuation
     of the cache withdrawal process when it is stopped pending destruction of
     all the objects.


 (*) Indicate negative lookup on an object:

	void fscache_object_lookup_negative(struct fscache_object *object);

     This is called to indicate to FS-Cache that a lookup process for an object
     found a negative result.

     This changes the state of an object to permit reads pending on lookup
     completion to go off and start fetching data from the netfs server as it's
     known at this point that there can't be any data in the cache.

     This may be called multiple times on an object.  Only the first call is
     significant - all subsequent calls are ignored.


 (*) Indicate an object has been obtained:

	void fscache_obtained_object(struct fscache_object *object);

     This is called to indicate to FS-Cache that a lookup process for an object
     produced a positive result, or that an object was created.  This should
     only be called once for any particular object.

     This changes the state of an object to indicate:
        (1) if no call to fscache_object_lookup_negative() has been made on
            this object, that there may be data available, and that reads can
            now go and look for it; and

        (2) that writes may now proceed against this object.


 (*) Indicate that object lookup failed:

        void fscache_object_lookup_error(struct fscache_object *object);

     This marks an object as having encountered a fatal error (usually EIO)
     and causes it to move into a state whereby it will be withdrawn as soon
     as possible.


 (*) Get and release references on a retrieval record:

        void fscache_get_retrieval(struct fscache_retrieval *op);
        void fscache_put_retrieval(struct fscache_retrieval *op);

     These two functions are used to retain a retrieval record whilst doing
     asynchronous data retrieval and block allocation.


 (*) Enqueue a retrieval record for processing.

        void fscache_enqueue_retrieval(struct fscache_retrieval *op);

     This enqueues a retrieval record for processing by the FS-Cache thread
     pool.  One of the threads in the pool will invoke the retrieval record's
     op->op.processor callback function.  This function may be called from
     within the callback function.


 (*) List of object state names:

        const char *fscache_object_states[];

     For debugging purposes, this may be used to turn the state that an object
     is in into a text string for display purposes.
```