FS-CACHE NETWORK FILESYSTEM API

There's an API by which a network filesystem can make use of the FS-Cache facilities. This is based around a number of principles:

(1) Caches can store a number of different object types. There are two main object types: indices and files. The first is a special type used by FS-Cache to make finding objects faster and to make retiring of groups of objects easier.

(2) Every index, file or other object is represented by a cookie. This cookie may or may not have anything associated with it, but the netfs doesn't need to care.

(3) Barring the top-level index (one entry per cached netfs), the index hierarchy for each netfs is structured according the whim of the netfs.

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

This document contains the following sections:

NETWORK FILESYSTEM DEFINITION

FS-Cache needs a description of the network filesystem. This is specified using a record of the following structure:

```
struct fscache_netfs {
        uint32_t                        version;
        const char                      *name;
        struct fscache_cookie           *primary_index;
        ...
};
```

This first two fields should be filled in before registration, and the third

will be filled in by the registration function; any other fields should just be ignored and are for internal use only.

The fields are:

 (1) The name of the netfs (used as the key in the toplevel index).

 (2) The version of the netfs (if the name matches but the version doesn't, the entire in-cache hierarchy for this netfs will be scrapped and begun afresh).

 (3) The cookie representing the primary index will be allocated according to another parameter passed into the registration function.

For example, kAFS (linux/fs/afs/) uses the following definitions to describe itself:

```
	struct fscache_netfs afs_cache_netfs = {
		.version	= 0,
		.name		= "afs",
	};
```

==========
INDEX DEFINITION
==========

Indices are used for two purposes:

 (1) To aid the finding of a file based on a series of keys (such as AFS's "cell", "volume ID", "vnode ID").

 (2) To make it easier to discard a subset of all the files cached based around a particular key - for instance to mirror the removal of an AFS volume.

However, since it's unlikely that any two netfs's are going to want to define their index hierarchies in quite the same way, FS-Cache tries to impose as few restraints as possible on how an index is structured and where it is placed in the tree. The netfs can even mix indices and data files at the same level, but it's not recommended.

Each index entry consists of a key of indeterminate length plus some auxilliary data, also of indeterminate length.

There are some limits on indices:

 (1) Any index containing non-index objects should be restricted to a single cache. Any such objects created within an index will be created in the first cache only. The cache in which an index is created can be controlled by cache tags (see below).

 (2) The entry data must be atomically journallable, so it is limited to about 400 bytes at present. At least 400 bytes will be available.

 (3) The depth of the index tree should be judged with care as the search function is recursive. Too many layers will run the kernel out of stack.

OBJECT DEFINITION

To define an object, a structure of the following type should be filled out:

```
struct fscache_cookie_def
{
        uint8_t name[16];
        uint8_t type;

        struct fscache_cache_tag *(*select_cache)(
                const void *parent_netfs_data,
                const void *cookie_netfs_data);

        uint16_t (*get_key)(const void *cookie_netfs_data,
                            void *buffer,
                            uint16_t bufmax);

        void (*get_attr)(const void *cookie_netfs_data,
                         uint64_t *size);

        uint16_t (*get_aux)(const void *cookie_netfs_data,
                            void *buffer,
                            uint16_t bufmax);

        enum fscache_checkaux (*check_aux)(void *cookie_netfs_data,
                                           const void *data,
                                           uint16_t datalen);

        void (*get_context)(void *cookie_netfs_data, void *context);

        void (*put_context)(void *cookie_netfs_data, void *context);

        void (*mark_pages_cached)(void *cookie_netfs_data,
                                  struct address_space *mapping,
                                  struct pagevec *cached_pvec);

        void (*now_uncached)(void *cookie_netfs_data);
};
```

This has the following fields:

(1) The type of the object [mandatory].

    This is one of the following values:

    (*) FSCACHE_COOKIE_TYPE_INDEX

        This defines an index, which is a special FS-Cache type.

    (*) FSCACHE_COOKIE_TYPE_DATAFILE

        This defines an ordinary data file.

(*) Any other value between 2 and 255

This defines an extraordinary object such as an XATTR.

(2) The name of the object type (NUL terminated unless all 16 chars are used) [optional].

(3) A function to select the cache in which to store an index [optional].

This function is invoked when an index needs to be instantiated in a cache during the instantiation of a non-index object. Only the immediate index parent for the non-index object will be queried. Any indices above that in the hierarchy may be stored in multiple caches. This function does not need to be supplied for any non-index object or any index that will only have index children.

If this function is not supplied or if it returns NULL then the first cache in the parent's list will be chosen, or failing that, the first cache in the master list.

(4) A function to retrieve an object's key from the netfs [mandatory].

This function will be called with the netfs data that was passed to the cookie acquisition function and the maximum length of key data that it may provide. It should write the required key data into the given buffer and return the quantity it wrote.

(5) A function to retrieve attribute data from the netfs [optional].

This function will be called with the netfs data that was passed to the cookie acquisition function. It should return the size of the file if this is a data file. The size may be used to govern how much cache must be reserved for this file in the cache.

If the function is absent, a file size of 0 is assumed.

(6) A function to retrieve auxilliary data from the netfs [optional].

This function will be called with the netfs data that was passed to the cookie acquisition function and the maximum length of auxilliary data that it may provide. It should write the auxilliary data into the given buffer and return the quantity it wrote.

If this function is absent, the auxilliary data length will be set to 0.

The length of the auxilliary data buffer may be dependent on the key length. A netfs mustn't rely on being able to provide more than 400 bytes for both.

(7) A function to check the auxilliary data [optional].

This function will be called to check that a match found in the cache for this object is valid. For instance with AFS it could check the auxilliary data against the data version number returned by the server to determine whether the index entry in a cache is still valid.

If this function is absent, it will be assumed that matching objects in a cache are always valid.

If present, the function should return one of the following values:

(*) FSCACHE_CHECKAUX_OKAY - the entry is okay as is
(*) FSCACHE_CHECKAUX_NEEDS_UPDATE - the entry requires update
(*) FSCACHE_CHECKAUX_OBSOLETE - the entry should be deleted

This function can also be used to extract data from the auxilliary data in the cache and copy it into the netfs's structures.

(8) A pair of functions to manage contexts for the completion callback [optional].

The cache read/write functions are passed a context which is then passed to the I/O completion callback function. To ensure this context remains valid until after the I/O completion is called, two functions may be provided: one to get an extra reference on the context, and one to drop a reference to it.

If the context is not used or is a type of object that won't go out of scope, then these functions are not required. These functions are not required for indices as indices may not contain data. These functions may be called in interrupt context and so may not sleep.

(9) A function to mark a page as retaining cache metadata [optional].

This is called by the cache to indicate that it is retaining in-memory information for this page and that the netfs should uncache the page when it has finished. This does not indicate whether there's data on the disk or not. Note that several pages at once may be presented for marking.

The PG_fscache bit is set on the pages before this function would be called, so the function need not be provided if this is sufficient.

This function is not required for indices as they're not permitted data.

(10) A function to unmark all the pages retaining cache metadata [mandatory].

This is called by FS-Cache to indicate that a backing store is being unbound from a cookie and that all the marks on the pages should be cleared to prevent confusion. Note that the cache will have torn down all its tracking information so that the pages don't need to be explicitly uncached.

This function is not required for indices as they're not permitted data.

==========
NETWORK FILESYSTEM (UN)REGISTRATION
==========

The first step is to declare the network filesystem to the cache. This also involves specifying the layout of the primary index (for AFS, this would be the

"cell" level).

The registration function is:

```
int fscache_register_netfs(struct fscache_netfs *netfs);
```

It just takes a pointer to the netfs definition. It returns 0 or an error as appropriate.

For kAFS, registration is done as follows:

```
ret = fscache_register_netfs(&afs_cache_netfs);
```

The last step is, of course, unregistration:

```
void fscache_unregister_netfs(struct fscache_netfs *netfs);
```

CACHE TAG LOOKUP

FS-Cache permits the use of more than one cache. To permit particular index subtrees to be bound to particular caches, the second step is to look up cache representation tags. This step is optional; it can be left entirely up to FS-Cache as to which cache should be used. The problem with doing that is that FS-Cache will always pick the first cache that was registered.

To get the representation for a named tag:

```
struct fscache_cache_tag *fscache_lookup_cache_tag(const char *name);
```

This takes a text string as the name and returns a representation of a tag. It will never return an error. It may return a dummy tag, however, if it runs out of memory; this will inhibit caching with this tag.

Any representation so obtained must be released by passing it to this function:

```
void fscache_release_cache_tag(struct fscache_cache_tag *tag);
```

The tag will be retrieved by FS-Cache when it calls the object definition operation select_cache().

INDEX REGISTRATION

The third step is to inform FS-Cache about part of an index hierarchy that can be used to locate files. This is done by requesting a cookie for each index in the path to the file:

```
struct fscache_cookie *
fscache_acquire_cookie(struct fscache_cookie *parent,
                       const struct fscache_object_def *def,
                       void *netfs_data);
```

This function creates an index entry in the index represented by parent, filling in the index entry by calling the operations pointed to by def.

Note that this function never returns an error - all errors are handled internally. It may, however, return NULL to indicate no cookie. It is quite acceptable to pass this token back to this function as the parent to another acquisition (or even to the relinquish cookie, read page and write page functions - see below).

Note also that no indices are actually created in a cache until a non-index object needs to be created somewhere down the hierarchy. Furthermore, an index may be created in several different caches independently at different times. This is all handled transparently, and the netfs doesn't see any of it.

For example, with AFS, a cell would be added to the primary index. This index entry would have a dependent inode containing a volume location index for the volume mappings within this cell:

```
	cell->cache =
		fscache_acquire_cookie(afs_cache_netfs.primary_index,
				       &afs_cell_cache_index_def,
				       cell);
```

Then when a volume location was accessed, it would be entered into the cell's index and an inode would be allocated that acts as a volume type and hash chain combination:

```
	vlocation->cache =
		fscache_acquire_cookie(cell->cache,
				       &afs_vlocation_cache_index_def,
				       vlocation);
```

And then a particular flavour of volume (R/O for example) could be added to that index, creating another index for vnodes (AFS inode equivalents):

```
	volume->cache =
		fscache_acquire_cookie(vlocation->cache,
				       &afs_volume_cache_index_def,
				       volume);
```

DATA FILE REGISTRATION

The fourth step is to request a data file be created in the cache. This is identical to index cookie acquisition. The only difference is that the type in the object definition should be something other than index type.

```
	vnode->cache =
		fscache_acquire_cookie(volume->cache,
				       &afs_vnode_cache_object_def,
				       vnode);
```

MISCELLANEOUS OBJECT REGISTRATION

An optional step is to request an object of miscellaneous type be created in the cache. This is almost identical to index cookie acquisition. The only difference is that the type in the object definition should be something other than index type. Whilst the parent object could be an index, it's more likely it would be some other type of object such as a data file.

```
xattr->cache =
        fscache_acquire_cookie(vnode->cache,
                               &afs_xattr_cache_object_def,
                               xattr);
```

Miscellaneous objects might be used to store extended attributes or directory entries for example.

SETTING THE DATA FILE SIZE

The fifth step is to set the physical attributes of the file, such as its size. This doesn't automatically reserve any space in the cache, but permits the cache to adjust its metadata for data tracking appropriately:

```
int fscache_attr_changed(struct fscache_cookie *cookie);
```

The cache will return -ENOBUFS if there is no backing cache or if there is no space to allocate any extra metadata required in the cache. The attributes will be accessed with the get_attr() cookie definition operation.

Note that attempts to read or write data pages in the cache over this size may be rebuffed with -ENOBUFS.

This operation schedules an attribute adjustment to happen asynchronously at some point in the future, and as such, it may happen after the function returns to the caller. The attribute adjustment excludes read and write operations.

PAGE READ/ALLOC/WRITE

And the sixth step is to store and retrieve pages in the cache. There are three functions that are used to do this.

Note:

(1) A page should not be re-read or re-allocated without uncaching it first.

(2) A read or allocated page must be uncached when the netfs page is released from the pagecache.

(3) A page should only be written to the cache if previous read or allocated.

This permits the cache to maintain its page tracking in proper order.

PAGE READ

Firstly, the netfs should ask FS-Cache to examine the caches and read the contents cached for a particular page of a particular file if present, or else allocate space to store the contents if not:

```
typedef
void (*fscache_rw_complete_t)(struct page *page,
                              void *context,
                              int error);

int fscache_read_or_alloc_page(struct fscache_cookie *cookie,
                               struct page *page,
                               fscache_rw_complete_t end_io_func,
                               void *context,
                               gfp_t gfp);
```

The cookie argument must specify a cookie for an object that isn't an index, the page specified will have the data loaded into it (and is also used to specify the page number), and the gfp argument is used to control how any memory allocations made are satisfied.

If the cookie indicates the inode is not cached:

(1) The function will return -ENOBUFS.

Else if there's a copy of the page resident in the cache:

(1) The mark_pages_cached() cookie operation will be called on that page.

(2) The function will submit a request to read the data from the cache's backing device directly into the page specified.

(3) The function will return 0.

(4) When the read is complete, end_io_func() will be invoked with:

    (*) The netfs data supplied when the cookie was created.

    (*) The page descriptor.

    (*) The context argument passed to the above function. This will be maintained with the get_context/put_context functions mentioned above.

    (*) An argument that's 0 on success or negative for an error code.

    If an error occurs, it should be assumed that the page contains no usable data.

    end_io_func() will be called in process context if the read is results in an error, but it might be called in interrupt context if the read is

successful.

Otherwise, if there's not a copy available in cache, but the cache may be able to store the page:

(1) The mark_pages_cached() cookie operation will be called on that page.

(2) A block may be reserved in the cache and attached to the object at the appropriate place.

(3) The function will return -ENODATA.

This function may also return -ENOMEM or -EINTR, in which case it won't have read any data from the cache.

PAGE ALLOCATE

Alternatively, if there's not expected to be any data in the cache for a page because the file has been extended, a block can simply be allocated instead:

```
int fscache_alloc_page(struct fscache_cookie *cookie,
                       struct page *page,
                       gfp_t gfp);
```

This is similar to the fscache_read_or_alloc_page() function, except that it never reads from the cache. It will return 0 if a block has been allocated, rather than -ENODATA as the other would. One or the other must be performed before writing to the cache.

The mark_pages_cached() cookie operation will be called on the page if successful.

PAGE WRITE

Secondly, if the netfs changes the contents of the page (either due to an initial download or if a user performs a write), then the page should be written back to the cache:

```
int fscache_write_page(struct fscache_cookie *cookie,
                       struct page *page,
                       gfp_t gfp);
```

The cookie argument must specify a data file cookie, the page specified should contain the data to be written (and is also used to specify the page number), and the gfp argument is used to control how any memory allocations made are satisfied.

The page must have first been read or allocated successfully and must not have been uncached before writing is performed.

If the cookie indicates the inode is not cached then:

(1) The function will return -ENOBUFS.

Else if space can be allocated in the cache to hold this page:

(1) PG_fscache_write will be set on the page.

(2) The function will submit a request to write the data to cache's backing device directly from the page specified.

(3) The function will return 0.

(4) When the write is complete PG_fscache_write is cleared on the page and anyone waiting for that bit will be woken up.

Else if there's no space available in the cache, -ENOBUFS will be returned. It is also possible for the PG_fscache_write bit to be cleared when no write took place if unforeseen circumstances arose (such as a disk error).

Writing takes place asynchronously.

MULTIPLE PAGE READ

A facility is provided to read several pages at once, as requested by the readpages() address space operation:

```
int fscache_read_or_alloc_pages(struct fscache_cookie *cookie,
                                struct address_space *mapping,
                                struct list_head *pages,
                                int *nr_pages,
                                fscache_rw_complete_t end_io_func,
                                void *context,
                                gfp_t gfp);
```

This works in a similar way to fscache_read_or_alloc_page(), except:

(1) Any page it can retrieve data for is removed from pages and nr_pages and dispatched for reading to the disk. Reads of adjacent pages on disk may be merged for greater efficiency.

(2) The mark_pages_cached() cookie operation will be called on several pages at once if they're being read or allocated.

(3) If there was an general error, then that error will be returned.

    Else if some pages couldn't be allocated or read, then -ENOBUFS will be returned.

    Else if some pages couldn't be read but were allocated, then -ENODATA will be returned.

    Otherwise, if all pages had reads dispatched, then 0 will be returned, the list will be empty and *nr_pages will be 0.

(4) end_io_func will be called once for each page being read as the reads

complete. It will be called in process context if error != 0, but it may be called in interrupt context if there is no error.

Note that a return of -ENODATA, -ENOBUFS or any other error does not preclude some of the pages being read and some being allocated. Those pages will have been marked appropriately and will need uncaching.

==========
PAGE UNCACHING
==========

To uncache a page, this function should be called:

```
void fscache_uncache_page(struct fscache_cookie *cookie,
                          struct page *page);
```

This function permits the cache to release any in-memory representation it might be holding for this netfs page. This function must be called once for each page on which the read or write page functions above have been called to make sure the cache's in-memory tracking information gets torn down.

Note that pages can't be explicitly deleted from the a data file. The whole data file must be retired (see the relinquish cookie function below).

Furthermore, note that this does not cancel the asynchronous read or write operation started by the read/alloc and write functions, so the page invalidation functions must use:

```
bool fscache_check_page_write(struct fscache_cookie *cookie,
                              struct page *page);
```

to see if a page is being written to the cache, and:

```
void fscache_wait_on_page_write(struct fscache_cookie *cookie,
                                struct page *page);
```

to wait for it to finish if it is.

When releasepage() is being implemented, a special FS-Cache function exists to manage the heuristics of coping with vmscan trying to eject pages, which may conflict with the cache trying to write pages to the cache (which may itself need to allocate memory):

```
bool fscache_maybe_release_page(struct fscache_cookie *cookie,
                                struct page *page,
                                gfp_t gfp);
```

This takes the netfs cookie, and the page and gfp arguments as supplied to releasepage(). It will return false if the page cannot be released yet for some reason and if it returns true, the page has been uncached and can now be released.

To make a page available for release, this function may wait for an outstanding storage request to complete, or it may attempt to cancel the storage request -

in which case the page will not be stored in the cache this time.

==========
INDEX AND DATA FILE UPDATE
==========

To request an update of the index data for an index or other object, the following function should be called:

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

This function will refer back to the netfs_data pointer stored in the cookie by the acquisition function to obtain the data to write into each revised index entry. The update method in the parent index definition will be called to transfer the data.

Note that partial updates may happen automatically at other times, such as when data blocks are added to a data file object.

==========
MISCELLANEOUS COOKIE OPERATIONS
==========

There are a number of operations that can be used to control cookies:

(*) Cookie pinning:

```
int fscache_pin_cookie(struct fscache_cookie *cookie);
void fscache_unpin_cookie(struct fscache_cookie *cookie);
```

These operations permit data cookies to be pinned into the cache and to have the pinning removed. They are not permitted on index cookies.

The pinning function will return 0 if successful, -ENOBUFS in the cookie isn't backed by a cache, -EOPNOTSUPP if the cache doesn't support pinning, -ENOSPC if there isn't enough space to honour the operation, -ENOMEM or -EIO if there's any other problem.

(*) Data space reservation:

```
int fscache_reserve_space(struct fscache_cookie *cookie, loff_t size);
```

This permits a netfs to request cache space be reserved to store up to the given amount of a file. It is permitted to ask for more than the current size of the file to allow for future file expansion.

If size is given as zero then the reservation will be cancelled.

The function will return 0 if successful, -ENOBUFS in the cookie isn't backed by a cache, -EOPNOTSUPP if the cache doesn't support reservations, -ENOSPC if there isn't enough space to honour the operation, -ENOMEM or -EIO if there's any other problem.

Note that this doesn't pin an object in a cache; it can still be culled to

make space if it's not in use.

COOKIE UNREGISTRATION

To get rid of a cookie, this function should be called.

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

If retire is non-zero, then the object will be marked for recycling, and all copies of it will be removed from all active caches in which it is present. Not only that but all child objects will also be retired.

If retire is zero, then the object may be available again when next the acquisition function is called. Retirement here will overrule the pinning on a cookie.

One very important note - relinquish must NOT be called for a cookie unless all the cookies for "child" indices, objects and pages have been relinquished first.

INDEX AND DATA FILE INVALIDATION

There is no direct way to invalidate an index subtree or a data file. To do this, the caller should relinquish and retire the cookie they have, and then acquire a new one.

FS-CACHE SPECIFIC PAGE FLAG

FS-Cache makes use of a page flag, PG_private_2, for its own purpose. This is given the alternative name PG_fscache.

PG_fscache is used to indicate that the page is known by the cache, and that the cache must be informed if the page is going to go away. It's an indication to the netfs that the cache has an interest in this page, where an interest may be a pointer to it, resources allocated or reserved for it, or I/O in progress upon it.

The netfs can use this information in methods such as releasepage() to determine whether it needs to uncache a page or update it.

Furthermore, if this bit is set, releasepage() and invalidatepage() operations will be called on a page to get rid of it, even if PG_private is not set. This allows caching to attempted on a page before read_cache_pages() to be called after fscache_read_or_alloc_pages() as the former will try and release pages it was given under certain circumstances.

This bit does not overlap with such as PG_private. This means that FS-Cache can be used with a filesystem that uses the block buffering code.

There are a number of operations defined on this flag:

```
int PageFsCache(struct page *page);
void SetPageFsCache(struct page *page)
void ClearPageFsCache(struct page *page)
int TestSetPageFsCache(struct page *page)
int TestClearPageFsCache(struct page *page)
```

These functions are bit test, bit set, bit clear, bit test and set and bit test and clear operations on PG_fscache.