Network Filesystem Helper Library

Overview

The network filesystem helper library is a set of functions designed to aid a network filesystem in implementing VM/VFS operations. For the moment, that just includes turning various VM buffered read operations into requests to read from the server. The helper library, however, can also interpose other services, such as local caching or local data encryption.

Note that the library module doesn't link against local caching directly, so access must be provided by the netfs.

Per-Inode Context

The network filesystem helper library needs a place to store a bit of state for its use on each netfs inode it is helping to manage. To this end, a context structure is defined:

```
struct netfs_i_context {
        const struct netfs_request_ops *ops;
        struct fscache_cookie   *cache;
};
```

A network filesystem that wants to use netfs lib must place one of these directly after the VFS `struct inode` it allocates, usually as part of its own struct. This can be done in a way similar to the following:

```
struct my_inode {
        struct {
                /* These must be contiguous */
                struct inode            vfs_inode;
                struct netfs_i_context  netfs_ctx;
        };
        ...
};
```

This allows netfslib to find its state by simple offset from the inode pointer, thereby allowing the netfslib helper functions to be pointed to directly by the VFS/VM operation tables.

The structure contains the following fields:

- `ops`

  The set of operations provided by the network filesystem to netfslib.

- `cache`

  Local caching cookie, or NULL if no caching is enabled. This field does not exist if fscache is disabled.

Inode Context Helper Functions

To help deal with the per-inode context, a number helper functions are provided. Firstly, a function to perform basic initialisation on a context and set the operations table pointer:

```
void netfs_i_context_init(struct inode *inode,
                          const struct netfs_request_ops *ops);
```

then two functions to cast between the VFS inode structure and the netfs context:

```
struct netfs_i_context *netfs_i_context(struct inode *inode);
struct inode *netfs_inode(struct netfs_i_context *ctx);
```

and finally, a function to get the cache cookie pointer from the context attached to an inode (or NULL if fscache is disabled):

```
struct fscache_cookie *netfs_i_cookie(struct inode *inode);
```

Buffered Read Helpers

The library provides a set of read helpers that handle the ->readpage(), ->readahead() and much of the ->write_begin() VM operations and translate them into a common call framework.

The following services are provided:

- Handle folios that span multiple pages.
- Insulate the netfs from VM interface changes.
- Allow the netfs to arbitrarily split reads up into pieces, even ones that don't match folio sizes or folio alignments and that may cross folios.
- Allow the netfs to expand a readahead request in both directions to meet its needs.

- Allow the netfs to partially fulfil a read, which will then be resubmitted.
- Handle local caching, allowing cached data and server-read data to be interleaved for a single request.
- Handle clearing of bufferage that aren't on the server.
- Handle retrying of reads that failed, switching reads from the cache to the server as necessary.
- In the future, this is a place that other services can be performed, such as local encryption of data to be stored remotely or in the cache.

From the network filesystem, the helpers require a table of operations. This includes a mandatory method to issue a read operation along with a number of optional methods.

Read Helper Functions

Three read helpers are provided:

```
void netfs_readahead(struct readahead_control *ractl);
int netfs_readpage(struct file *file,
                   struct page *page);
int netfs_write_begin(struct file *file,
                      struct address_space *mapping,
                      loff_t pos,
                      unsigned int len,
                      unsigned int flags,
                      struct folio **_folio,
                      void **_fsdata);
```

Each corresponds to a VM address space operation. These operations use the state in the per-inode context.

For ->readahead() and ->readpage(), the network filesystem just point directly at the corresponding read helper; whereas for ->write_begin(), it may be a little more complicated as the network filesystem might want to flush conflicting writes or track dirty data and needs to put the acquired folio if an error occurs after calling the helper.

The helpers manage the read request, calling back into the network filesystem through the suppplied table of operations. Waits will be performed as necessary before returning for helpers that are meant to be synchronous.

If an error occurs and netfs_priv is non-NULL, ops->cleanup() will be called to deal with it. If some parts of the request are in progress when an error occurs, the request will get partially completed if sufficient data is read.

Additionally, there is:

```
* void netfs_subreq_terminated(struct netfs_io_subrequest *subreq,
                               ssize_t transferred_or_error,
                               bool was_async);
```

which should be called to complete a read subrequest. This is given the number of bytes transferred or a negative error code, plus a flag indicating whether the operation was asynchronous (ie. whether the follow-on processing can be done in the current context, given this may involve sleeping).

Read Helper Structures

The read helpers make use of a couple of structures to maintain the state of the read. The first is a structure that manages a read request as a whole:

```
struct netfs_io_request {
        struct inode            *inode;
        struct address_space    *mapping;
        struct netfs_cache_resources cache_resources;
        void                    *netfs_priv;
        loff_t                  start;
        size_t                  len;
        loff_t                  i_size;
        const struct netfs_request_ops *netfs_ops;
        unsigned int            debug_id;
        ...
};
```

The above fields are the ones the netfs can use. They are:

- `inode`
- `mapping`

  The inode and the address space of the file being read from. The mapping may or may not point to inode->i_data.

- `cache_resources`

  Resources for the local cache to use, if present.

- `netfs_priv`

The network filesystem's private data. The value for this can be passed in to the helper functions or set during the request. The ->cleanup() op will be called if this is non-NULL at the end.

- `start`
- `len`

  The file position of the start of the read request and the length. These may be altered by the ->expand_readahead() op.

- `i_size`

  The size of the file at the start of the request.

- `netfs_ops`

  A pointer to the operation table. The value for this is passed into the helper functions.

- `debug_id`

  A number allocated to this operation that can be displayed in trace lines for reference.

The second structure is used to manage individual slices of the overall read request:

```
struct netfs_io_subrequest {
        struct netfs_io_request *rreq;
        loff_t                  start;
        size_t                  len;
        size_t                  transferred;
        unsigned long           flags;
        unsigned short          debug_index;
        ...
};
```

Each subrequest is expected to access a single source, though the helpers will handle falling back from one source type to another. The members are:

- `rreq`

  A pointer to the read request.

- `start`
- `len`

  The file position of the start of this slice of the read request and the length.

- `transferred`

  The amount of data transferred so far of the length of this slice. The network filesystem or cache should start the operation this far into the slice. If a short read occurs, the helpers will call again, having updated this to reflect the amount read so far.

- `flags`

  Flags pertaining to the read. There are two of interest to the filesystem or cache:

  - `NETFS_SREQ_CLEAR_TAIL`

    This can be set to indicate that the remainder of the slice, from transferred to len, should be cleared.

  - `NETFS_SREQ_SEEK_DATA_READ`

    This is a hint to the cache that it might want to try skipping ahead to the next data (ie. using SEEK_DATA).

- `debug_index`

  A number allocated to this slice that can be displayed in trace lines for reference.

Read Helper Operations

The network filesystem must provide the read helpers with a table of operations through which it can issue requests and negotiate:

```
struct netfs_request_ops {
        void (*init_request)(struct netfs_io_request *rreq, struct file *file);
        int (*begin_cache_operation)(struct netfs_io_request *rreq);
        void (*expand_readahead)(struct netfs_io_request *rreq);
        bool (*clamp_length)(struct netfs_io_subrequest *subreq);
        void (*issue_read)(struct netfs_io_subrequest *subreq);
        bool (*is_still_valid)(struct netfs_io_request *rreq);
        int (*check_write_begin)(struct file *file, loff_t pos, unsigned len,
                                 struct folio *folio, void **_fsdata);
        void (*done)(struct netfs_io_request *rreq);
        void (*cleanup)(struct address_space *mapping, void *netfs_priv);
};
```

The operations are as follows:

- `init_request()`

  [Optional] This is called to initialise the request structure. It is given the file for reference and can modify the ->netfs_priv value.

- `begin_cache_operation()`

  [Optional] This is called to ask the network filesystem to call into the cache (if present) to initialise the caching state for this read. The netfs library module cannot access the cache directly, so the cache should call something like fscache_begin_read_operation() to do this.

  The cache gets to store its state in ->cache_resources and must set a table of operations of its own there (though of a different type).

  This should return 0 on success and an error code otherwise. If an error is reported, the operation may proceed anyway, just without local caching (only out of memory and interruption errors cause failure here).

- `expand_readahead()`

  [Optional] This is called to allow the filesystem to expand the size of a readahead read request. The filesystem gets to expand the request in both directions, though it's not permitted to reduce it as the numbers may represent an allocation already made. If local caching is enabled, it gets to expand the request first.

  Expansion is communicated by changing ->start and ->len in the request structure. Note that if any change is made, ->len must be increased by at least as much as ->start is reduced.

- `clamp_length()`

  [Optional] This is called to allow the filesystem to reduce the size of a subrequest. The filesystem can use this, for example, to chop up a request that has to be split across multiple servers or to put multiple reads in flight.

  This should return 0 on success and an error code on error.

- `issue_read()`

  [Required] The helpers use this to dispatch a subrequest to the server for reading. In the subrequest, ->start, ->len and ->transferred indicate what data should be read from the server.

  There is no return value; the netfs_subreq_terminated() function should be called to indicate whether or not the operation succeeded and how much data it transferred. The filesystem also should not deal with setting folios uptodate, unlocking them or dropping their refs - the helpers need to deal with this as they have to coordinate with copying to the local cache.

  Note that the helpers have the folios locked, but not pinned. It is possible to use the ITER_XARRAY iov iterator to refer to the range of the inode that is being operated upon without the need to allocate large bvec tables.

- `is_still_valid()`

  [Optional] This is called to find out if the data just read from the local cache is still valid. It should return true if it is still valid and false if not. If it's not still valid, it will be reread from the server.

- `check_write_begin()`

  [Optional] This is called from the netfs_write_begin() helper once it has allocated/grabbed the folio to be modified to allow the filesystem to flush conflicting state before allowing it to be modified.

  It should return 0 if everything is now fine, -EAGAIN if the folio should be regrabbed and any other error code to abort the operation.

- `done`

  [Optional] This is called after the folios in the request have all been unlocked (and marked uptodate if applicable).

- `cleanup`

  [Optional] This is called as the request is being deallocated so that the filesystem can clean up ->netfs_priv.

Read Helper Procedure

The read helpers work by the following general procedure:

- Set up the request.
- For readahead, allow the local cache and then the network filesystem to propose expansions to the read request. This is then proposed to the VM. If the VM cannot fully perform the expansion, a partially expanded read will be performed, though this may not get written to the cache in its entirety.
- Loop around slicing chunks off of the request to form subrequests:
  - If a local cache is present, it gets to do the slicing, otherwise the helpers just try to generate maximal slices.
  - The network filesystem gets to clamp the size of each slice if it is to be the source. This allows rsize and

    chunking to be implemented.
  - The helpers issue a read from the cache or a read from the server or just clears the slice as appropriate.
  - The next slice begins at the end of the last one.
  - As slices finish being read, they terminate.
- When all the subrequests have terminated, the subrequests are assessed and any that are short or have failed are reissued:
  - Failed cache requests are issued against the server instead.
  - Failed server requests just fail.
  - Short reads against either source will be reissued against that source provided they have transferred some more data:
    - The cache may need to skip holes that it can't do DIO from.
    - If NETFS_SREQ_CLEAR_TAIL was set, a short read will be cleared to the end of the slice instead of reissuing.
- Once the data is read, the folios that have been fully read/cleared:
  - Will be marked uptodate.
  - If a cache is present, will be marked with PG_fscache.
  - Unlocked
- Any folios that need writing to the cache will then have DIO writes issued.
- Synchronous operations will wait for reading to be complete.
- Writes to the cache will proceed asynchronously and the folios will have the PG_fscache mark removed when that completes.
- The request structures will be cleaned up when everything has completed.

Read Helper Cache API

When implementing a local cache to be used by the read helpers, two things are required: some way for the network filesystem to initialise the caching for a read request and a table of operations for the helpers to call.

The network filesystem's ->begin_cache_operation() method is called to set up a cache and this must call into the cache to do the work. If using fscache, for example, the cache would call:

```
int fscache_begin_read_operation(struct netfs_io_request *rreq,
                                 struct fscache_cookie *cookie);
```

passing in the request pointer and the cookie corresponding to the file.

The netfs_io_request object contains a place for the cache to hang its state:

```
struct netfs_cache_resources {
        const struct netfs_cache_ops    *ops;
        void                            *cache_priv;
        void                            *cache_priv2;
};
```

This contains an operations table pointer and two private pointers. The operation table looks like the following:

```
struct netfs_cache_ops {
        void (*end_operation)(struct netfs_cache_resources *cres);

        void (*expand_readahead)(struct netfs_cache_resources *cres,
                                 loff_t *_start, size_t *_len, loff_t i_size);

        enum netfs_io_source (*prepare_read)(struct netfs_io_subrequest *subreq,
                                             loff_t i_size);

        int (*read)(struct netfs_cache_resources *cres,
                    loff_t start_pos,
                    struct iov_iter *iter,
                    bool seek_data,
                    netfs_io_terminated_t term_func,
                    void *term_func_priv);

        int (*prepare_write)(struct netfs_cache_resources *cres,
                             loff_t *_start, size_t *_len, loff_t i_size,
                             bool no_space_allocated_yet);

        int (*write)(struct netfs_cache_resources *cres,
                     loff_t start_pos,
                     struct iov_iter *iter,
                     netfs_io_terminated_t term_func,
                     void *term_func_priv);

        int (*query_occupancy)(struct netfs_cache_resources *cres,
                               loff_t start, size_t len, size_t granularity,
                               loff_t *_data_start, size_t *_data_len);
};
```

With a termination handler function pointer:

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

The methods defined in the table are:

- `end_operation()`

  [Required] Called to clean up the resources at the end of the read request.

- `expand_readahead()`

  [Optional] Called at the beginning of a netfs_readahead() operation to allow the cache to expand a request in either direction. This allows the cache to size the request appropriately for the cache granularity.

  The function is passed poiners to the start and length in its parameters, plus the size of the file for reference, and adjusts the start and length appropriately. It should return one of:

  - `NETFS_FILL_WITH_ZEROES`
  - `NETFS_DOWNLOAD_FROM_SERVER`
  - `NETFS_READ_FROM_CACHE`
  - `NETFS_INVALID_READ`

  to indicate whether the slice should just be cleared or whether it should be downloaded from the server or read from the cache - or whether slicing should be given up at the current point.

- `prepare_read()`

  [Required] Called to configure the next slice of a request. ->start and ->len in the subrequest indicate where and how big the next slice can be; the cache gets to reduce the length to match its granularity requirements.

- `read()`

  [Required] Called to read from the cache. The start file offset is given along with an iterator to read to, which gives the length also. It can be given a hint requesting that it seek forward from that start position for data.

  Also provided is a pointer to a termination handler function and private data to pass to that function. The termination function should be called with the number of bytes transferred or an error code, plus a flag indicating whether the termination is definitely happening in the caller's context.

- `prepare_write()`

  [Required] Called to prepare a write to the cache to take place. This involves checking to see whether the cache has sufficient space to honour the write. `*_start` and `*_len` indicate the region to be written; the region can be shrunk or it can be expanded to a page boundary either way as necessary to align for direct I/O. i_size holds the size of the object and is provided for reference. no_space_allocated_yet is set to true if the caller is certain that no data has been written to that region - for example if it tried to do a read from there already.

- `write()`

  [Required] Called to write to the cache. The start file offset is given along with an iterator to write from, which gives the length also.

  Also provided is a pointer to a termination handler function and private data to pass to that function. The termination function should be called with the number of bytes transferred or an error code, plus a flag indicating whether the termination is definitely happening in the caller's context.

- `query_occupancy()`

  [Required] Called to find out where the next piece of data is within a particular region of the cache. The start and length of the region to be queried are passed in, along with the granularity to which the answer needs to be aligned. The function passes back the start and length of the data, if any, available within that region. Note that there may be a hole at the front.

  It returns 0 if some data was found, -ENODATA if there was no usable data within the region or -ENOBUFS if there is no caching on this file.

Note that these methods are passed a pointer to the cache resource structure, not the read request structure as they could be used in other situations where there isn't a read request structure as well, such as writing dirty data to the cache.

API Function Reference

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]netfs_library.rst`, line 607)

Unknown directive type "kernel-doc".

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]netfs_library.rst`, line 608)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: fs/netfs/buffered_read.c
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]netfs_library.rst`, line 609)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: fs/netfs/io.c
```