RFC: Recursive Exclusive / Shared Lock for Bypass VOL

(Draft)

John Mainzer (john.mainzer@lifeboat.llc)

The Bypass VOL distinguishes between multi-thread safe and non multi-thread safe operations, and executes them in parallel or sequentially accordingly.

To enforce the required shared or exclusive access, the Bypass VOL requires a recursive exclusive / shared lock. The lock must be recursive because HDF5 allows user callbacks that may call the HDF5 library recursively, and because concurrently executing threads in the Bypass VOL may have to access the native VOL.

Functionally, the recursive exclusive / shared lock is largely identical to a recursive R/W lock. However, referring to it as such will be confusing in the context of the Bypass VOL.

1 Introduction 3
Functional Requirements for the Recursive Exclusive / Shared Lock 3
1.1 Recursive Lock 3
1.2 Precedence Policy 4
1.3 Portability 4
2 Design of the Recursive Exclusive / Shared Lock 4
2.1 Support for Recursive Locks 5
2.2 Support for Multiple Precedence Policies 5
2.3 Support for Flush when Shifting from Exclusive to Shared Access 5
3 Implementation Details 5
3.1 P-threads Implementation 6
3.2 Windows Implementation 10
4 Testing 10
5 Recommendation 10
1. Acknowledgements 10
2. Revision History 11

1 Introduction

The Bypass VOL leverages multi-thread VOL support to execute a subset of HDF5 data set I/O requests concurrently. It does this by querying the native VOL for the relevant metadata, and then performing the actual I/O directly. While the metadata lookups must be done sequentially, the I/O can be done in parallel and can overlap with metadata lookups required to service other requests and their associated I/O. Thus it is possible to have multiple I/O requests in progress simultaneously, allowing the bypass VOL to make full use of the available I/O bandwidth. Requests that can't be handled concurrently are simply relayed to the native VOL as serial operations.

Managing a mix of operations that can either run concurrently with other operations of the same class, or that require exclusive access to the underlying native VOL connector requires a recursive exclusive / shared lock. This lock must be recursive, since HDF5 API calls that require the exclusive lock may invoke user callbacks that may invoke other HDF5 API calls. Further, the concurrent operations must access the native VOL to lookup the necessary metadata. Since these calls pass through H5VL and the Bypass VOL before reaching the native VOL, their access to the native VOL must also be managed with recursive shared locks.

This RFC details the proposed design of a recursive exclusive / shared lock for use in the Bypass VOL.

Functional Requirements for the Recursive Exclusive / Shared Lock

The exclusive / shared lock is just a variant on a read / write lock. It is called an exclusive / shared lock in the Bypass VOL as we will perform both raw data reads and writes under the shared lock. Thus calling it a read / write lock would be confusing.

Read / Write locks are a well known[1] synchronization construct. In a nutshell, a R/W lock allows either an arbitrary number of reader threads, or a single writer thread into a critical region. Similarly, the exclusive / shared lock will allow an arbitrary number of selected operations to execute under the shared lock, or a single instance of all other operations to execute under the exclusive lock. The major difference is that the exclusive / shared lock must flush the underlying HDF5 file(s) after executing one or more exclusive operations before allowing shared access.

While P-threads supports R/W locks, we can't use the P-threads R/W lock as the basis of the exclusive / shared lock due to portability requirements, and the recursion requirement which the P-thread R/W doesn't support.

1.1 Recursive Lock

A recursive lock is simply a lock that allows a thread that already holds the lock to successful request the lock again. The lock function must detect this case, increment a thread specific recursive reference count and return success. Similarly, the unlock function must verify that the calling thread holds the lock, decrement the thread specific recursive reference count, and drop the lock if the reference count has dropped to zero.

[1] Volume 2 of "UNIX Network Programming" by W. Richard Stevens contains a good discussion of R/W locks in chapter 8.

The exclusive / shared lock must be recursive, since some HDF5 API calls allow the user to specify callbacks into user code. Since user code can call the HDF5 library in these callbacks, recursive locks are required to avoid deadlocks.

The obvious use case is in non multi-thread safe operations. However, as currently planned, thread with the shared lock will make calls into the HDF5 library to obtain the locations of the target raw data. These calls will pass through the H5VL code and then re-enter the Bypass VOL before being re-directed into the native VOL. Such threads must be identified and allowed back into the Bypass VOL with the shared lock even though they would normally require the exclusive lock.

1.2 Precedence Policy

Typically, R/W locks give access precedence to writers. Thus in the absence of write locks, readers are allowed immediate access. Once a writer requests access, no further read locks are granted, and the writer is blocked until all read locks are dropped, at which point the writer receives the exclusive write lock. All lock requests received while a write lock request is either pending or active are blocked until the write lock is dropped. If one or more of the pending requests is a write lock request, one of the pending writers is given access. If not, all read lock requests are granted.

While the above policy is a reasonable starting point, the exclusive / shared lock should be implemented with a configurable precedence policy. Obvious options are:

- Exclusive locks have precedence
- Shared locks have precedence
- Exclusive locks have precedence, but allow any pending shared access requests access after each exclusive request completes even if there is also a pending exclusive access request.

There is no need to implement different precedence policies to begin with. It is enough to create the infrastructure and implement one policy.

While it should not be considered for the initial implementation, there is also the possibility of threads dropping the shared lock after all necessary metadata has been obtained from the native VOL. This raises the possibility of allowing operations requiring exclusive access to proceed while file I/O is in progress. This will have to be approached with care, as it raises the possibility of collisions – for example, the deletion of a dataset out from under an ongoing read or write. That said, there may be some performance gains here.

1.3 Portability

In its final version, the recursive exclusive / shared lock must support at least these P-threads and Windows. If C11 becomes widely supported in Windows compilers, a C11 implementation may be sufficient.

The initial version need only support P-threads.

2 Design of the Recursive Exclusive / Shared Lock

W. Richard Stevens presents a R/W lock implementation in chapter 8 of Volume 2 of “UNIX Network Programming”. This outline of the recursive Exclusive / Shared lock design uses his implementation as

a baseline, and restricts itself discussing the changes necessary to support recursive locks and other features required by the design.

2.1 Support for Recursive Locks

To implement the recursive feature, we must maintain a recursive reference count for each thread that currently holds a lock (be it shared or exclusive), so that we can drop that thread's lock when the number of un-locks equals the number of locks. In addition to the reference count, we also track whether the initial lock was a shared lock or an exclusive lock.

In a regular R/W lock, this data would be used to reject write lock requests from threads that already hold a read lock, and vise-versa.

In the context of the exclusive / shared lock for the Bypass VOL, we use this data to convert exclusive lock requests to shared lock requests if the thread already holds a shared lock, and shared lock requests to exclusive lock requests if the thread already holds the exclusive lock. This sounds dangerous in the first case, but since such recursive lock requests will only be generated by Bypass VOL code, with care it should be safe enough.

For P-threads, do this by associating a key with the recursive exclusive / shared lock.

When a thread makes an initial lock request, allocate a structure containing the type of lock (exclusive or shared) and the recursive lock reference count, and associate it with the thread via pthread_setspecific().

On subsequent lock or unlock requests, a pointer to this structure is obtained via pthread_getspecific(), and the reference count is either incremented or decremented.

If the reference count reaches zero, the lock is dropped, the structure is discarded, and pthread_setspecific() is used to set the thread specific value associated with the key back to NULL.

Initial lock requests are detected when pthread_getspecific() returns NULL.

2.2 Support for Multiple Precedence Policies

Support for multiple precedence policies can be implemented by adding a policy ID parameter to the recursive exclusive / shared lock initialization routine. This policy ID is stored in the recursive exclusive / shared lock structure, and consulted whenever policy relevant decisions are made.

2.3 Support for Flush when Shifting from Exclusive to Shared Access

Similarly, support the requirement that the underlying HDF5 file(s) be flushed when shifting from a single thread with exclusive access to one or more threads with shared access by adding a function pointer to the recursive exclusive / shared lock initialization routine. This function pointer must be invoked whenever a thread with exclusive access drops its lock, and one or more threads are allowed to obtain shared locks. Note that this function must complete before any thread is allowed access.

3 Implementation Details

While the above recursive exclusive / shared lock design section should have provided a conceptual overview of the the recursive exclusive / shared lock, for efficient maintenance a discussion of code structure and location is also needed.

The location of code for the recursive exclusive / shared lock is TBD at this time, as is the appropriate naming convention.

Details of the various recursive exclusive / shared lock implementations are discussed in the following sections.

3.1 P-threads Implementation

The main structure for the P-threads recursive exclusive / shared lock implementation is located in ???.h, and reproduced below with its header comment.

```
/*****************************************************************************
 *
 * Structure ???_pt_rec_xs_lock_t
 *
 * An exclusive / shared lock is a lock that allows either an arbitrary number
 * of threads shared access to the protected critical region, or a single
 * thread exclusive access.  Note that such locks are usually referred to
 * as read / write locks.  However, given the application of the exclusive /
 * shared lock in the Bypass VOL, using the typical name would be confusing.
 *
 * A recursive lock is one that allows a thread that already has a lock (be it
 * exclusive or shared) to successfully request the lock again, only dropping
 * the lock when the number of un-lock calls equals the number of lock calls.
 *
 * The management of recursive locks in the Bypass VOL is a bit peculiar.
 * While initial exclusive and shared lock request must be granted as such,
 * recursive requests for either type of lock must be converted to the type of
 * lock already held by the requesting thread.
 *
 * To see this, consider the following cases:
 *
 * 1) Suppose a thread that currently holds the exclusive lock executes a
 *    a user callback that invokes a HDF5 API that is usually executed under
 *    a shared lock in the Bypass VOL.  Obviously, this API call must be
 *    executed under the currently held exclusive lock to avoid an immediate
 *    deadlock.
 *
 * 2) Now suppose that a thread with a shared lock executing in the Bypass VOL
 *    has to make a call into the native VOL - as is currently done to obtain
 *    the necessary metadata to perform I/O requests.  These calls have to
 *    be executed under the global mutex and thus would normally be executed
 *    with the exclusive lock.  Assuming such calls into the native VOL are
 *    routed through H5VL and then back through the Bypass VOL, they would
 *    normally be executed under the exclusive lock - which would cause an
 *    immediate deadlock.
 *
 * Finally, note that we can't use the p-threads R/W lock as the base of the
 * exclusive / shared lock, as while it permits recursive read locks, it
 * disallows recursive write locks.
 *
 * This structure is a catchall for the fields needed to implement a
 * p-threads based recursive exclusive / shared lock, and for the
 * associated statistics collection fields.
 *
 * This recursive exclusive / shared lock implementation is an extension of
```

```
 * the R/W lock implementation given in "UNIX network programming" Volume 2,
 * Chapter 8 by w. Richard Stevens, 2nd edition.
 *
 * Individual fields are discussed below.
 *
 *                                          JRM  -- 4/17/25
 *
 * tag:         Unsigned 32 bit integer field used for sanity checking.  This
 *              field must always be set to ???_PT_REC_XS_LOCK_TAG.
 *              If this structure is allocated dynamically, remember to set
 *              it to some invalid value before discarding the structure.
 *
 * policy       Integer containing a code indicating the precedence policy
 *              used by the exclusive / shared lock.  The supported policies
 *              are listed below:
 *
 *              ???__XS_LOCK_POLICY__FAVOR_EXCLUSIVE_ACCESS:
 *
 *              If selected, the exclusive / shared lock will grant a
 *              pending exclusive lock request if there are both pending
 *              shared and exclusive lock requests.
 *
 *
 *              --- Define other policies here ---
 *
 *
 * mutex:       Mutex used to maintain mutual exclusion on the fields of
 *              of this structure.
 *
 * shared_cv:  Condition variable used for threads waiting for shared access.
 *
 * exclusive_cv: Condition variable used for threads waiting for exclusive
 *              access.
 *
 * waiting_shared_count: 32 bit integer used to maintain a count of the number
 *              of threads waiting for shared access.  This value should always
 *              be non-negative.
 *
 * waiting_exclusive_count: 32 bit integer used to maintain a count of the
 *              number of threads waiting for exclusive access.  This value
 *              should always be non-negative.
 *
 * The following two fields could be combined into a single field, with
 * the count of threads with shared access being represented by a positive
 * value, and the number of threads with exclusive access by a negative value.
 * Two fields are used to facilitate sanity checking.
 *
 * active_shared: 32 bit integer used to maintain a count of the number of
 *              threads that currently hold a shared lock.  This value
 *              must be zero if active_exclusive is positive. It should
 *              never be negative.
 *
 * active_exclusive: 32 bit integer used to maintain a count of the number of
 *              threads that currently hold an exclusive lock.  This value
 *              must always be either 0 or 1, and must be zero if
 *              active_shared is positive.  It should never be negative.
 *
 * rec_entry_count_key: Instance of pthread_key_t used to maintain
```

```
 *              a thread specific lock type and recursive entry count
 *              for all threads holding a lock.
 *
 * x2s_func:    Pointer to a function that must be called when a thread with
 *              an exclusive lock drops it, and one or more other threads
 *              obtain the shared lock.  Note that this function must complete
 *              before any thread is given the shared lock.
 *
 * x2s_data:    Void pointer that is passed to x2s_func when it is invoked.
 *
 * stats:       Instance of ???_pt_rec_xs_lock_stats_t used to track
 *              statistics on the recursive exclusive / shared lock.  See
 *              the declaration of the structure for discussion of its fields.
 *
 *              Note that the stats are gathered into a structure because
 *              we must obtain the mutex when reading the statistics to
 *              avoid changes while the statistics are being read.  Collecting
 *              them into a structure facilitates this.
 *
 ***********************************************************************/

#define ???_PT_REC_XS_LOCK_TAG                      0XABCD

#define ???__XS_LOCK_POLICY__FAVOR_EXCLUSIVE_ACCESS 0

typedef herr_t (* ???_pt_rec_xs_x2s_func_t)(void * data);

typedef struct ???_pt_rec_xs_lock_t {

    uint32_t                            tag;
    int32_t                             policy;
    pthread_mutex_t                     mutex;
    pthread_cond_t                      shared_cv;
    pthread_cond_t                      exclusive_cv;
    int32_t                             waiting_shared_count;
    int32_t                             waiting_exclusive_count;
    int32_t                             active_shared;
    int32_t                             active_exclusive;
    pthread_key_t                       rec_entry_count_key;
    int32_t                             exclusive_rec_entry_count;
    ???_pt_rec_xs_x2s_func_t            x2s_func;
    void *                              x2s_data;
    struct ???_pt_rec_xs_lock_stats_t   stats;

} ???_pt_rec_xs_lock_t;
```

Readers familiar with Stevens will note that this structure is very similar to that used for the R/W lock discussed in chapter 8 of Volume 2 of “UNIX Network Programming”. The major differences being the additions of the `policy`, `rec_entry_count_key`, `x2s_func`, `x2s_data` and `stats` fields. The `stats` field is used heavily in testing the recursive exclusive / shared lock, and will likely be useful in other contexts as well.

The reference count for each thread currently holding either an exclusive or shared lock is maintained in an instance of `???_pt_rec_entry_count_t`. This structure is defined in ???.h, and reproduced below with its header comment.

```
/******************************************************************************
 *
 * Structure ???_pt_rec_entry_count_t
 *
 * Structure associated with the rec_entry_count_key defined in
 * ???_pt_rec_xs_lock_t.
 *
 * The primary purpose of this structure is to maintain a count of recursive
 * locks so that the lock can be dropped when the count drops to zero.
 *
 * Additional fields are included for purposes of sanity checking.
 *
 * Individual fields are discussed below.
 *
 *                                              JRM -- 4/17/25
 *
 * tag:         Unsigned 32 bit integer field used for sanity checking.  This
 *              fields must always be set to
 *              ???_PT_REC_XS_REC_ENTRY_COUNT_TAG, and should be set to
 *              some invalid value just before the structure is freed.
 *
 * exclusive_lock:  Boolean field that is set to TRUE if the count is for an
 *              exclusive lock, and to FALSE if it is for a shared lock.
 *
 * rec_lock_count: Count of the number of recursive lock calls, less
 *              the number of recursive unlock calls.  The lock in question
 *              is dropped when the count drops to zero.
 *
 ******************************************************************************/

#define ???_PT_REC_XS_REC_ENTRY_COUNT_TAG       0XABBA

typedef struct H5TS_pt_rec_entry_count_t {

    uint32_t    tag;
    hbool_t     exclusive_lock;
    int64_t     rec_lock_count;

} ???_pt_rec_entry_count_t;
```

The definition of ???_pt_rec_xs_lock_stats_t is also located in ???.h, along with the numerous macros used to maintain it. As it is used purely for record keeping, the structure is not reproduced here.

The functions that operate the recursive exclusive / shared lock are located in ???.c. In addition to the obvious

1. shared lock
2. exclusive lock, and
3. unlock

functions, there are also functions to:

- setup and take down the lock,
- get, reset, and print the statistics, and
- allocate, initialize, and free instances of `???_pt_rec_entry_count_t`.

Note that there are no “try shared lock” or ”try exclusive lock” functions, as these are not required at present. Needless to say, they can be implemented when and if needed.

3.2 Windows Implementation

TBD

4 Testing

A common approach to testing a locking primitive is to write code to detect locking failures, create a large number of threads to randomly gain and drop the lock, and run it until it until either a failure is detected, or one has reasonable confidence in the primitive. Repeating this test on a variety of platforms with various core counts allows greater confidence.

The tests themselves create some number of threads that are directed to perform some specific number of exclusive and/or shared locks. Once a lock is gained, the recursive locking feature is tested by requiring the thread to perform a bounded drunken walk of locks and unlocks until the lock is dropped, and the thread proceeds to the next exclusive or shared lock request as directed.

Once all threads have completed, statistics collected from the threads and the recursive exclusive / shared lock proper are cross checked – the test fails if there is any discrepancy. The statistics from the lock are also examined to verify that no un-allowable conditions occurred (i.e. two writers at once). Finally, return codes from all locking functions are checked, and the test fails if any of these calls returns an unexpected failure.

With this context, the tests can be described as follows:

Basic – single thread that verifies basic functionality and error rejection.

Mob of shared access requests – many threads gain and drop shared locks.

Mob of exclusive access requests – many threads gain and drop exclusive locks.

Mixed mob – many threads gain and drop a mix of exclusive and shared locks.

5 Recommendation

Implement, test, and employ in the initial version of the multi-thread Bypass VOL.

Acknowledgements

This work is supported by the U.S. Department of Energy, Office of Science under award number DE-SC0023583 for Phase II SBIR project "Toward multi-threaded concurrency in HDF5”

Revision History

April 17, 2025: Version 1 circulated for comment within Lifeboat.