Solutions for lock ordering issues exposed by the current multi-threaded implementation of H5I (draft)

John Mainser
August 31, 2025

In our work, we encountered lock ordering issues resulting from the decision to use locks on individual IDs. This document outlines several approaches to eliminating those locks within the ID indexes maintained by the H5I code.

ID Locking Forced by the Possibility of Free Function Failure

Ideally, the free_func would be multi-thread safe, and be guaranteed to succeed. This would allow ID deletion without either a lock on the ID or acquisition of the global mutex[1]. Note however the hidden assumption that either frees of memory used to store ID values are ordered so as to prevent any thread from retaining a pointer to freed memory, or they are put on a free list and retained until the no threads retain pointers to them, and then freed or reallocated.

While it should not be a problem for newly developed multi-thread VOL connectors, requiring that the free_func() be guaranteed to succeed presents issues in HDF5 proper[2]. While we haven't investigated the matter in detail, the impression is that HDF5 uses this feature of the free_func() to control the order in which IDs and their associated data structures are discarded at library shutdown. As a free_func() that is guaranteed to succeed is essential for a fully lock free index, it will be useful to determine the level of effort required to attain this in the HDF5 library.

From initial investigation of this matter, it appears that requiring free functions to succeed in all cases would be very expensive. To avoid this cost, two possible workarounds have presented themselves that should be consistent with a fully lock free index.

free_func_will_succeed() function

The basic idea here is that if we can't guarantee that the free function will succeed, develop a test for when it will, and assume it will fail if this test fails.

For each index type whose free function cannot reasonably be made to succeed in all cases, add a free_func_will_succeed() function. This function must return TRUE if a subsequent call to the free function is guaranteed to succeed.

When this free_func_will_succeed() function is present, it is called whenever it is possible that the ID will be discarded and the free function will be called.

---

1 Since it would allow us to mark the ID as deleted, and then call the free_func() at our leisure.

2 This issue has become urgent, as integration of H5I with a thread safe version of H5VL has made it more or less impossible to maintain lock ordering between the lock on the ID, and the global mutex. At present, this problem is addressed by identifying potential deadlocks, and backing out of them. However, this is very inefficient, and adds a great deal of complexity.

If the free_func_will_succeed() function returns FALSE, the operation fails without calling the free function.

If it returns TRUE, the target ID is deleted, and the free function (which must succeed) is then called – thus allowing deletion of IDs without the need for locking.

We are currently investigating this option. While it is too early to say if this approach is workable, there is little doubt that it will require a significant level of effort. A further issue is that the solution requires changes to the H5I API.

Restrict Access to the Calling Thread if the Free Function Fails

The basic idea behind this approach flows from a conversation with the HDF5 developers. It was suggested that if a thread successfully reduces the reference count on an ID to zero, then absent programming errors, that thread should be the only thread able to access the target ID – making a lock on the ID unnecessary for purposes of the free function.

Unfortunately, this doesn't work, since all extant IDs are accessible to all threads via iteration. However, it did suggest an alternate approach that may be workable. In essence, the idea is to make the thread that reduces the reference count of an ID to zero responsible for any necessary repeated decrement reference count calls, and hide the target ID from all other threads. The revised processing for the decrement reference call is outlined below.

1. Lookup the instance of H5I_mt_id_info_t associated with the target ID. Return failure if the lookup fails. Otherwise do an atomic read of the kernel k[3]. Save the local image of the kernel in k1.

2. If k1.marked is true, or k1.closing is true, and either k1.tid_valid is false or k1.tid doesn't match the current thread, return failure.

3. Decrement k1.count in the local copy. If k1.count is zero, also set k1.closing to TRUE, k1.tid_valid to TRUE, and k1.tid to the id of the current thread.

4. Attempt to overwrite k with an atomic compare exchange strong. If this operation fails, return to 1.

5. If the new value of k.count is positive, return k.count.

6. If the new value of k.count is zero, call the free func on k.object.

7. If the call to the free_func() succeeds, return zero and delete the target instance of H5I_mt_id_info_t from from the hash table, and return success.

8. If the free func fails, load a fresh local copy of the kernel – call it k2. Verify that k2 equals k1. Set k2.count to 1, overwrite the global version with a compare exchange strong, and return failure.

---

3 Recall that the kernel of an instance of H5I_mt_id_info_t is an atomic structure containing all fields that

In addition to the above, all lookup code in H5I must be modified as per step 2 above.

In the context of the free function, this approach avoids explicit locking by effectively locking the target ID to the calling thread. This approach has a chance of working since initially at least, we plan to handle HDF5 library shutdown with a single thread, and HDF5 already has facilities for discarding orphan IDs.

Orphan IDs are a real possibility, as the iteration call prevents the deletion of entries out from under the iteration callback by wrapping it in inc ref / dec ref calls. Thus, it is possible that a dec ref that is supposed to be the final dec ref on the target will happen to hit between these calls, causing the final dec ref to be invoked by the iteration code instead, and transferring responsibility for the repeated dec ref calls to the iterating thread. If we can't think of a way to prevent this, we will need a mechanism to make all IDs visible to the single thread at shutdown.

ID Locking Forced by Future ID management

While the realize and discard callbacks for future IDs must be retained for the single thread and thread safe builds of the HDF5 library[4], we should look into alternate ways of providing the functionality in the multi-thread case. This is needed for the following reasons:

- To realize a future ID, at present the H5I code must:

    1. Call the realize callback to obtain the ID of the index entry containing the actual value.

    2. Remove the actual value ID from the index while retaining its object pointer.

    3. Call the discard callback on the object pointer associated with the future ID,

    4. Set the object pointer field associated with the future ID equal to the object pointer retained when removing the actual ID and then mark the ID as real.

    While the failure of the realize callback can be handled gracefully, inability to remove the actual ID, or failure of the discard callback leaves the index and client in a failure state with no obvious path for recovery or graceful error handling.

- Even if the above issues are resolved, the current API requires a lock on the ID, as there is no provision for rolling back the operation if the ID is modified between the beginning and end of future ID resolution.

- Finally, the realize callback can, and sometimes does, stall pending availability of the required data.

In a nutshell, the current future ID mechanism forces a level of coupling between the index and the client that complicates the H5I significantly and makes it impossible to avoid locking at least the ID

4 Or perhaps not – if multi-thread H5I is not merged into the HDF5 library until the next major version.

even if we assume that the realize callback, the removal of the real ID, and the discard callback are multi-thread safe and always succeed.

Further, the realize callback can (and does in some cases) stall pending computation of the value for the future ID. This allows the client to stall and possibly deadlock operations on the host index – another strong argument for redesigning the interface. Note that this is done to allow API calls to stall pending resolution of future IDs, and thus it is a feature not a bug. Thus, we must maintain this functionality without the deficits of the current implementation.

Fortunately, this is not an issue that has to be dealt with immediately. To our knowledge, only two VOL connectors used the future ID mechanism. Since both use the existing interface with either the thread safe or the single thread build, the point is moot until some VOL that requires some sort of future ID capability tries to run multi-thread using the multi-thread updates to the service packages in the HDF5 library[5].

That said, the issue has to be resolved eventually. To begin the conversation, we offer the following straw man replacement that would allow H5I to deal with future IDs in atomic operations without the issues outlined above.

```
typedef herr_t H5I_progress_func_t(hid_t id);

hid_t H5Ireserve_future_id(H5I_type_t type, H5I_progress_func_t progress_cb);

herr_t H5Idefine_future_id(H5I_type_t type, hid_t id, void * object);
```

Here, H5Ireserve_future_id() would create a future ID in the target type or index, and return this ID to the caller. This reserved ID would remain undefined until its value was defined by a subsequent call to H5Idefine_future_id().

The H5Idefine_future_id() call would set the void pointer on the associated future ID to the supplied value, mark the future ID as being defined, and, if the progress callback is NULL, signal the index's future ID condition variable. The call will fail if the supplied future ID no longer exists, or if it has already been defined.

Attempts to access a future ID will be handled as follows, depending on whether the progress_cb is NULL.

1. If the future ID is defined (i.e. its object pointer is not NULL and it is marked as being defined), return the object pointer.

2. If the future ID does not exist (i.e. it has been deleted from the index) return NULL.

3. If the future ID exists, is undefined and the progress_cb is not NULL, call the progress callback, and go to 1 when it returns.

4. If the future ID exists, is undefined and the progress_cb is NULL wait on the index's future ID condition variable. Go to 1 when the condition variable is signaled.

---

5 That is, the H5E, H5I, H5P, H5CX, H5VL, and eventually H5S and probably H5T modules. While we want to make H5FD multi-thread safe as well, this is for reasons largely unrelated to VOL connectors.

While the above is a straw man, the basic idea is to remove the index as much as possible from the process of realizing the future ID, and thus from the associated error management and synchronization issues.