MT issues with HDF5 Future IDs and Possible Solutions

Integrating Multi-Thread Safe H5I into the HDF5 Library

At the conceptual level, integrating a multi-thread safe module (H5I in this case) into the HDF5 library is trivial – just start with the thread safe build, and push the global mutex that prevents multiple threads from entering the HDF5 library down below the multi-thread safe module. To do this, we must remove the lock / unlock calls to the global mutex from the H5I public API calls, and wrap all calls out of H5I to other, non multi-thread safe HDF5 packages in global mutex lock / unlock calls.

While this sounds trivial, in practice, one immediately runs into difficulties stemming from the design philosophy differences between lock free multi-thread and single thread programming. Specifically, while it may allow cleanup after the fact, lock free multi-thread wants to perform data structure modifications atomically, with the option of re-trying if another thread changes the playing field while the first thread is getting ready for its atomic operation. In contrast, single thread can allow data structures to be temporarily inconsistent, as long as everything is tidied up by the time an operation completes.

To see this, consider the operations that can be performed on the data associated with individual entries in an index. As listed above, they include:

1. Insert
2. Delete (typically when ref count drops to zero, but not always)
3. Increment ref count
4. Decrement ref count
5. Modify void pointer associated with an ID
6. Convert from future to real ID

For H5I to function with multiple threads active in it simultaneously, all these operations must be performed atomically. However, items 2, and 4 involve the free_func callback that was specified when the index was created, and 6 requires invocation of the realize and delete callbacks that were specified when the future ID was inserted into the index.

From the above, it is clear that, at least in the context of the HDF5 library proper, we must wrap these callbacks in lock / unlock calls to the global mutex. However, this is only the beginning of the problem. Generally speaking, these call backs are not guaranteed to succeed, and cannot be rolled back if a lock free operation fails and has to be re-tried. Thus in addition to grabbing the global mutex, we must also lock the structure associated with the ID to ensure that we can complete the operation if the callback succeeds. Note that this lock must be recursive, as there is nothing to prevent the callback from accessing the index and ID that occasioned the callback.

The situation is more difficult in case 6 above (future ID), as there is no guarantee that both callbacks will succeed – potentially leaving the index and client in a corrupted state. Further, the realize callback is used to stall until the data necessary to realize the future ID becomes available in at least one VOL connector. While this objective is understandable, and the implementation is reasonable enough with single threaded or thread safe HDF5, it is a non-starter for multi-thread HDF5 VOL connectors. To see this, observe that both a lock on the individual ID and the global mutex must be held during this stall – resulting in performance issues at a minimum, and deadlocks at worst.

This leaves the question of how to address these issues.

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. While I haven't investigated the matter in detail, my 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.

While the realize and discard callbacks for future IDs must be retained for the single thread and thread safe builds of the HDF5 library[2], we should look into alternate ways of providing the functionality in the multi-thread case. This is needed for the following reasons:

- Even if both the realize and discard callbacks succeed, neither call can be rolled back – which requires a lock on the target ID.

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

  1. Call the realize callback to obtain the 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 fact that 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

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

2 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 fact that the realize callback can (and does in some cases) stall pending completion of the value for the future ID allows the client to stall and possibly deadlock operations on the host index is 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 is a feature not a bug – just not with the current implementation.

Fortunately, this is not an issue that has to be dealt with immediately. To my 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[3].

That said, the issue has to be resolved eventually. To begin the conversation, I 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 value of the object pointer.

2. If the future ID does not exists (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.

3 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.

A related issue is raised by the H5Iiterate() and H5Isearch() public API calls – both of which use H5I_iterate(). That function iterates through the target index, and call a possibly user supplied callback on each ID. Since there is no guarantee that this callback is thread safe, it must be wrapped in the global mutex. Further, to avoid the possibility of the target ID being deleted out from under the user supplied function, the target ID must be locked for the duration of the call.
Note that this is a case where recursive locks on IDs are exercised, as at least one iteration through an index or type is triggered by a delete callback in the existing HDF5 regression tests.

The danger of running user supplied callbacks while holding any sort of lock need not be repeated. Thus the plan is to replace the public H5Iiterate() and H5Isearch() API calls with a more generic API, that allows the user to iterate through the target index using a get first / get next mechanism, and thus avoid the necessity of holding any locks when the user operates on the IDs in the index. It then becomes the users responsibility to ensure that either the ID and its associated void pointer are not deleted out from under the user's operation on same, or that such cases are handled gracefully.

As the external API is not needed for the prototype, it hasn't been implemented yet. However the API will likely be as follows:

```
hid_t H5Iget_first(H5I_type_t type, hid_t *id_ptr, void ** object_ptr);

herr_t H5Iget_next(H5I_type_t type, hid_t old_id, hid_t *id_ptr, void ** object_ptr);
```

where H5Iget_first() returns the first ID in the target index in *id_ptr, and the pointer to the associated object in *object_ptr[4]. Similarly, H5Iget_next() returns the next ID in the target index after the supplied old_id in *id_ptr, and its associated object pointer in *object_ptr. This mechanism is implemented and used internally in H5I, and handles ID deletions and insertions during the iteration gracefully. If the old_id no longer exists, it just returns the next ID after the deleted old_id in hash order. Inserted IDs may or may not appear in the iteration, depending on whether the hash code of the inserted ID is greater or less than that of the old id in the next call to H5Iget_next().

The H5Iiterate() and H5Isearch() API calls will have to be disabled in the multi-thread build at some point. To see why, observe that once the future ID issue is dealt with, and the free_func() is required to always succeed, these functions will be the only driver for locking individual IDs in H5I. Similarly, if the free_func() can be made thread safe in all cases, we will be able to avoid taking the global mutex in H5I as well.

While the above callback and iteration issues are the major complication in integrating lock free, multi-thread safe modules into the existing HDF5 library, there is one other minor one. Specifically some of the calls in the func enter / func exit macros are not multi-thread safe.

One of these, H5CX (context) is already on the list of modules to me made multi-thread safe, so no worries bypassing it for now.

4 This pointer will probably have to be unwrapped, but this is a detail for later.

The other, H5CS (call stack) maintains a call stack for debugging purposes. From a brief examination of the code, it seems that if isn't already multi-thread safe, it can be made so easily. Again, this issue has been bypassed for now. I don't expect it to be a major issue, but it must either be made multi-thread safe if it isn't already, or be removed from the multi-thread build.