Making H5I Multi-Thread Safe:
Outline of the Problem and a Prototype Solution

John Mainzer
join.mainzer@lifeboat.llc

1 Introduction

The purpose of this document[1] is to record our design decisions and implementation approach as we proceed with the development of multi-threaded functionality in HDF5, in particular, making H5I module multi-thread safe. The implementation will doubtless have to be revised to address issues as they are encountered during further development of multi-threaded HDF5 functionality. Similarly, the implementation will require optimization before a production release of the software. The document will be updated accordingly based on the new findings and the changes to the implementation.

This document is organized as follows. In section 2 we describe the initial non-thread-safe H5I package. In section 3 we outline the issues to be addressed in making H5I multi-thread safe. Section 4 talks about design considerations, and section 5 discusses a current prototype implementation of the multi-threaded H5I module[2]. For convenience and reference purposes Appendices contain descriptions of public and internal H5I API calls, along with call trees, relevant structure definitions, and descriptions of their processing with particular emphasis on multi-thread safety issues.

2 Quick Overview of H5I

H5I exists to provide indexing services both to the HDF5 library proper, and to application programs. The basic services may be summarized as follows:

- Create and delete types of indexes. Here the type of an index indicates the type of entries the index supports. In the HDF5 library proper, types include error messages, files, datasets, etc.

- Insert, lookup, and delete entries in individual indexes. To insert an entry the user provides the type of the target index, and a void pointer to whatever data is to be associated with the entry, and receives an identifier (ID) in return. This ID is used for subsequent lookups and deletions. Note that IDs have reference counts, and under normal circumstances are not deleted until their reference counts drop to 0.

  The private API also provides calls to:

  - lookup the ID associated with a given void pointer,
  - modify the void pointer associated with an existing index entry
  - insert an entry into an index with a specified ID

[1] The document is an update of the earlier version (H5I-2022-09-18.pdf). The discussion of the initial implementation of a multi-thread safe version of H5I was added along with copy editing.
[2] The source code can be found on GitHub https://github.com/LifeboatLLC/Experimental.

- Iteration and searching through indexes. Here the user provides a function and an index type. The function is executed on every ID in the target index (in the case of iteration) or until it reports success (in the case of searches).

- Miscellaneous services that include incrementing and decrementing reference counts on entries and types of indexes, tests for validity, fetching the number of index entries of a given type, deletion of types of indexes with all their entries, etc.

Unfortunately, there are also miscellaneous calls to look up the file or name associated with entries in certain indexes – which result in dependencies on packages in addition to H5E. The hope is that these dependencies can be resolved either through re-architecting or through multi-thread safety requirements on the external calls – but we will not know until the target packages have been examined in greater detail.

The iteration calls also raise issues.

In iteration calls that pass the void pointer associated with the current ID to the iteration call back function, H5I calls H5I_unwrap on this pointer before passing it to the callback. While this is a no-op in the public API, for some internal types of indexes (H5I_FILE, H5I_GROUP, H5I_DATASET, H5I_ATTR, and H5I_DATATYPE), the call to H5I_unwrap() results in a H5VL call, and a subsequent call into the appropriate VOL connector. While I have largely bypassed this issue for now, it is an unknown that will have to be resolved before we proceed with further development.

The external API iteration calls (H5Isearch() and H5Iiterate()) also present issues, as the HDF5 library has no control over these user supplied call backs – in principle they can make arbitrary calls back into the HDF5 library.

3 Multi-Thread Issues in H5I

As with H5E, leaving aside the issue of the unexpected dependencies, there appear to be no fundamental reasons why H5I can't be made multi-thread safe. That said, there are a number of issues to be dealt with. Before discussing how these challenges might be addressed, it will be useful to discuss each of these issues in greater detail.

3.1 Use of other HDF5 packages by H5I

As should be obvious from the above outline of the H5I package, there is no functional reason why H5I has to make calls to packages in the HDF5 library other than H5E for error reporting. That said, in its current implementation it does – specifically it has calls to:

- H5MM
- H5FL,
- H5E
- H5VL

- H5F

Note that while H5P is included in H5I source code files, it appears to be used only for access to a single constant used in a H5VL call – and thus it isn't listed above.

As in H5E, H5MM and H5FL are easily avoided by using the C dynamic memory allocation functions directly, and by either not maintaining free lists, or maintaining them internally.

The dependency on H5E is a larger issue, as it presents the possibility of lock ordering issues since we may need to call H5E from H5I, which in turn may call H5I. In the long term, the safest way to resolve this would be to remove H5E's dependency on H5I. However, there are several other ways which would work – albeit with greater danger of inadvertent insertion of deadlocks. Perhaps the easiest would be to make any locks used by H5I recursive.

The dependencies on H5VL and H5F spring from the above mentioned public API calls to determine the file or name associated with certain types of IDs, and the calls to H5I__unwrap() in some of the iteration calls. Indeed, the problem is potentially greater than this, as initial call trees for these public API calls indicate that H5CX and H5T are also involved, along with calls into VOL connectors. That said, the current focus is on H5I proper – thus these issues are left as known unknowns for now.

The callbacks used in both the public and private iteration calls (H5Isearch, H5Iiterate(), H5I_iterate(), etc.) are of similar concern as they have the potential to introduce dependencies, and thus potential lock ordering issues.

For the immediate objective (retrofitting multi-thread safety on H5E, H5I, H5P, H5CX, and H5VL), this is not too troubling, as we need only consider H5I iteration calls from these five packages – and an initial scan indicates that only H5VL is involved. The remainder of the HDF5 library is not an immediate concern, as any H5I iteration calls from these packages will be protected by the global lock for now.

Finally, the callbacks used in the public iteration calls (H5Isearch() and H5Iiterate()) have the potential to introduce arbitrary indirect dependencies. In principle, the free_func provided to H5Iregister_type, and the realize and discard functions provided to H5Iregister_future() present similar issues.

3.2 Multi-thread thread issues in H5I proper

A review of the H5I public, private, and developer APIs reveals the following multi-thread safety issues in the current H5I implementation.

3.2.1 Use of uthash to implement indexes

H5I uses uthash – a collection of macros – to implement the hash tables used to implement indexes. According to uthash documentation, uthash can be made thread safe by wrapping all uthash macros in a read / write lock. Write locks are required for operations modifying the target hash table, with read locks being sufficient for all other operations.

At present, uthash is not integrated into the HDF5 error reporting system. This is convenient, as it removes H5E from the problem, and thus avoids any lock ordering / lock recursion issues from this quarter at least. However, if HDF5 is going to stay with uthash indefinitely, this oversight should be corrected.

3.2.2 Use of global variables

The list of types of indexes and the next available index type are kept in global variables – with the obvious potential for race conditions and resulting data corruption.

Locks around the appropriate critical regions are an obvious solution, but they present lock ordering issues in the event of failure due to the resulting H5E calls. Locks could be dropped prior to error calls, but a solution based on atomic operations would be preferable as it would likely be faster and less error prone.

3.2.3 Potential race conditions in indexes and index entries

A multi-thread version of H5I must allow simultaneous operations by multiple threads. Thus, it is possible that there will be simultaneous operations on a given index, or even on a given ID in that index.

The structures currently used to implement both types of indexes (H5I_type_info_t) and index entries proper (H5I_id_info_t) contain a number of fields where race conditions are an obvious issue – most notably reference counts, and (in the case of types of indexes) counts of existing IDs.

As with the global variables, locks around critical regions are the obvious solution, but solutions based on atomic operations would be preferable.

3.2.4 Mark and Sweep Operations

H5I allows scans and searches of types of indexes. My understanding is that to avoid breaking existing tests, H5I does not immediately delete entries deleted by the user during scans or searches. Instead, entries are marked for deletion, and then deleted in a subsequent sweep.

The same approach is used in the clear and destroy operations – which compound the problem by leaving index entries in a partially deleted state pending the subsequent sweep to complete the deletion.

This has the effect of making iteration operations of whatever type large critical regions – which is obviously undesirable in a multi-thread implementation. At a minimum, repairing this will require algorithmic changes, and may require subtle modifications to the H5I API as well.

3.2.5 Support for Future IDs

Addition of support for the asynchronous VOL required the addition of support for future IDs. In particular, there is code in the ID lookup function that attempts to convert a future ID into an actual ID. This operation is somewhat involved, and presents a variety of race conditions should the same future ID be looked up simultaneously by multiple threads.

The critical region here is sufficiently large that locking may be the only practical solution. (Note: future IDs are used by DAOS for asynchronous operations.)

3.2.6 Public API Race Conditions

The nature of an index service in a multi-thread environment makes it possible for the client application to create race conditions – for example, it will always be possible for one thread to delete an ID (or a whole index) out from under another thread.

This is an unsolvable problem from the perspective of H5I – thus H5I's responsibility will be to at least appear to execute operations in some order, and to keep indexes in an internally consistent state. It will be the responsibility of the client to either avoid race conditions of the above type, or to handle them gracefully.

4 Design Considerations

The unexpected dependencies of H5I on other packages are worrying. Even if my initial impressions are correct, and we can both avoid all potential deadlocks from all calls from H5I to other packages, and tame the public iteration call back problem through strict constraints on the functions, the resulting code will be error prone. Since the HDF5 library is under constant development, re-insertion of deadlocks is inevitable unless we have simple, easily followed and verified rules for avoiding them. Failure to do this will result in maintenance nightmare, as prompt detection of newly inserted deadlocks in regression testing is problematic at best.

With this point in mind, I can't but conclude that in the long term, a multi-thread safe implementation of H5I will have to hold no locks whenever it makes a call to another package – thus removing all H5I related lock ordering concerns[3]. Ideally, this would be accomplished by making the multi-thread safe implementation of H5I lock free, but small critical regions containing no function calls and protected by locks should be workable as well.

3 With the possible exception of H5E – if we remove that package's dependency on H5I.

However, in the short term, time and resource constraints give us a strong incentive to minimize the changes required for the initial implementation. This, however, must be balanced with efficiency and integration concerns, as not only must the packages required to support multi-thread VOL connectors be multi-thread safe, they must also work together without deadlocks, and must facilitate sufficient performance gains to justify their development.

While it would be best to avoid locks completely to sidestep the lock ordering question, there are several impediments:

- For thread safety, uthash requires a R/W lock around all of its macros. Thus, avoiding locks here would require replacing uthash with some other lock free data structure. Note that this is not as large a problem as it may seem, as until we integrate uthash into HDF5's error reporting system, these locks present no potential for deadlocks. Further, if we do integrate uthash properly, we should be able to drop locks before calling H5E to report errors.

- Many of the calls that modify indexes modify the relevant data structures in a variety of places – both in the data structures and the code. Making all these changes atomically while maintaining ordering constraints would likely require significant redesign of the code, and may be impossible without public API changes.

- The mark and sweep algorithm used in H5Iclear_type() and H5Idestroy_type() leave the target index in an invalid state between the mark and sweep passes. Absent algorithm changes, it will be difficult to prevent access to the target index during this period without locking.

For these reasons, the approach to making H5E multi-thread safe offered below is largely lock based, and presumes that the external dependencies can be managed well enough to make this design approach viable at least in the short term.

At best, I regard this approach to be an initial solution aimed at facilitating relatively rapid implementation of a prototype. At worst, it will prove impractical in the light of further investigation, and will have to be re-worked in part or in whole. Its primary advantage is simplicity. It is offered only in rough sketch form, as there is no point in putting further effort into it until the full requirements are well understood.

5 Making H5I Multi-Thread Safe

Given that resource considerations dictate at least a partially lock based solution for the first cut, the least expensive approach would be to put a recursive lock on H5I, allowing only one thread into H5I at a time. The recursive lock would be needed to allow for calls from H5E back into H5I.

However, this allows for no concurrency – and thus brings into question the performance requirement. Fortunately, with minimal effort, we should be able to relax this one thread in H5I at a time constraint as follows:

5.1 Global Variables

H5I uses two global variables to maintain a list of indexes (or types as they are called in the code). These are H5I_type_info_array_g, which contains pointers to the instances of H5I_type_info_t that are the root structures for each index or type, and H5I_next_type_g, which is an integer containing the index of the next free slot in H5I_type_info_t.

The operations on these global variables are both simple and tightly coupled. Thus, with minimal code changes, we should be able to maintain the consistency of these variables with atomic operations and not have to resort to locks.

The code that searches for unused entries in H5I_type_info_array_g does present issues. The obvious solution is to simply delete it, but this functionality can probably be retained via the definition of a special null pointer indicating that a cell is allocated but not valid.

5.2 Indexes (AKA Types)

The base structure for each index (or type) is an instance of H5I_type_info_t.[4] After initialization is complete, the instance of H5I_type_info_t associated with each index (or type) is pointed to by H5I_type_info_array_t[type] – here type is the integer ID of the index. NULL entries in this array indicate that the associated index has not yet been created or has been discarded.

Individual entries in each index are represented by instances of H5I_id_info_t. Each index entry is assigned a unique ID. This ID has the ID of the host index (or type) encoded in it – allowing an ID to be looked up even if the ID of the index it resides in is not available.

When a new ID is registered, an instance of H5I_id_info_t is allocated, initialized, and loaded with the supplied void *. An ID is allocated in the target index (or type), and the new instance of H5I_id_info_t is inserted into the target indexes hash table, with the new ID as the key.

Thus, each index (or type) is represented with one instance of H5I_type_info_t, with zero or more instances of H5I_id_info_t stored in the hash table that is rooted in the instance of H5I_type_info_t.

To sketch out a locking protocol, we need to catalog the fields that are modified during the various operation on indexes – this is provided in Appendix 3.

Review of this table shows that API calls that don't modify the hash table modify few fields in the relevant instances of H5I_type_info_t and H5I_id_info_t – usually just one but occasionally two.

This suggests the following approach to ensuring multi-thread safety in the indexes (or types) proper:

- Associated a recursive read/write lock with each index (or type).

4 See Appendix 1 for the definition of this and all other structures mentioned in this section.

- Require all H5I API calls that modify the hash table to acquire a write lock on the target index (or type) before proceeding.

- Require all other H5I API calls to obtain a read lock on the target index (or type). Further, identify all fields of H5I_type_info_t and H5I_id_info_t modified by these calls and require them to be accessed via atomics.

This approach is clearly simple and relatively easy to implement, and provides the locking required to use uthash in a multi-thread environment. Further, if future IDs are excluded[5], it offers a significant degree of concurrency. However, there are a few points that should be discussed.

The first is efficacy – will this approach maintain the indexes in an internally consistent state, and make it appear to the clients that the H5I API calls have been executed in some sequential order? The only issue I see here is the modification of fields in H5I_type_info_t and H5I_id_info_t while only a read lock is held.

If only one field is modified, and that modification is done atomically, that should preserve the appearance of sequential execution of API calls. With two or more fields, there is the possibility of interleaving – however the only externally visible example of this is the regular and application reference counts in H5I_id_info_t. Since reference count decrements present the possibility of deletion, those operations will require a write lock. Thus, only increments will be done concurrently – and it is hard to see how allowing the regular and application reference counts to be temporarily out of sync will be an issue as long as both remain positive. That said, this is a point to keep in mind as I continue to review the target packages.

A second point is the observation that by creating a separate read / write lock for each index, I have introduced a potential deadlock. This potential could be realized if any of the user callbacks supplied to the iteration API calls, or any of the external calls that I have left un-investigated attempts an H5I API call on another index. This seems improbable but not impossible – I must keep an eye out for it.

Third, note that all the iteration API calls will require a write lock – severely restricting concurrency. While this constraint could be relaxed selectively, the proposed solution is at best a temporary measure – and thus I don't see the point.

Finally, the use of recursive read/write locks should allow H5I to keep its error calls unchanged. While I haven't investigated in detail, I expect that the recursive locks will also be necessary for some iterative APIs.

5 If an index contains a future id, a simple lookup of that id will trigger an attempt to convert it into an actual id. Along with other things, this operation requires a deletion from the hash table – and thus a write lock. Fortunately, future objects are rare, and we could modify indexes to track the number of future objects they contain – permitting use of a read lock in most cases.

6 The Implementation of Multi-Thread Safe H5I

6.1 Conceptual Overview

At the most basic level, the objective is to create a multi-thread safe implementation of H5I, and integrate it into the HDF5 library in such a fashion that multiple threads can be active in H5I, but without allowing multiple threads into non-multi-thread safe sections of the HDF5 library – or into non-multi-thread safe VOL connectors.

While there is some interaction between these goals at the implementation level, conceptually, they are unrelated – and thus are dealt with separately to the extent possible.

6.1.1 Multi-Thread Safe H5I at the Conceptual Level

At its core, the function of H5I is to create and maintain indices. In the multi-thread environment, this functionality can be implemented either a locking[6] or lock free based approach. While the lock free route is typically more complex, and introduces significant memory management problems in languages that don't support garbage collection, it has two major advantages.

First, by eliminating (or more correctly in this case, minimizing) locks, one eliminates (or minimizes) opportunities for deadlocks. While this shouldn't be a major issue when developing from scratch, or modifying a well-documented code base, experience in implementing a multi-thread safe version of H5I has underlined the fact that HDF5 has many odd twists and turns that could easily trip up a more rigid lock-based approach.

Second, the lock free approach facilitates a more flexible approach to synchronization, where we can concentrate on maintaining the consistency of HDF5 data structures, and leave higher level synchronization concerns to the user program – much as we do in the HDF5 MPI build. Thus, while HDF5 must maintain consistency of its data structures and impose some sequential order on the effects of concurrent operations, all other synchronization concerns must be handled by the host program. Thus, for example, if the host program needs to avoid modifications to an index during an iteration through it, it must enforce the required mutual exclusion.

Before proceeding, it is worth noting that the usual reasons for selecting a lock free design are performance and scalability. While flexibility and avoiding deadlocks were the driving factors in its selection in this case, it will be interesting to see if we get dividends from these directions as well.

The decision to use a lock free design to the extent possible[7] presented three major technical hurdles to be overcome:

6 For example, we could use uthash with its macros wrapped in read or write locks as appropriate to implement a multi-thread safe version of the basic indexing function in H5I.

7 As shall be seen, the need to inter-operate with the remainder of the HDF5 library and VOL connectors that are not thread safe makes some locking unavoidable.

1. Selection and implementation of some sort of lock free indexing mechanism.
2. Lock free management of the structures used to manage IDs in an index.
3. Garbage collection.

The first two are obvious enough, but the garbage collection issue may need explanation for those not familiar with lock free algorithms and data structures. Briefly, since multiple threads may be accessing a given data structure simultaneously, it is possible for a structure in the data structure to be removed from the data structure while other threads still retain pointers to it.

With careful use of atomic variables, this is easy enough to manage. However, there is still the problem of how to determine when no thread retains a pointer to the discarded structure so that it can be released to the heap. In languages that support it, garbage collection handles this problem nicely – albeit at significant cost.

In the absence of garbage collection, alternate solutions must be found. To date, placing discarded structures on type specific lock free queues, and using type specific heuristics to determine when entries on these free lists may be safely reallocated or returned to the heap seems to work well. Indeed, given my (admittedly dated) experience with the performance issues surrounding garbage collection, it may be preferable in time critical applications.

There were also some minor technical issues surrounding the management of indexes and lists thereof. As shall be seen when we discuss implementation details, use of atomic variables makes these relatively straight forward.

6.1.2 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 tries 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, a 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, 4, and 6 all involve callbacks that were specified when the index was created (in the case of Delete and Decrement ref count), or when the ID was inserted into the index (in the case of converting the ID from a future ID to a real ID).

From the above, it is clear that 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 callbacks are not guaranteed to succeed, and cannot be rolled back if a lock free operation fails and has to be retried. Thus, in addition to grabbing the global mutex, we must also lock the structure associated with the ID[8] to ensure that we can complete the operation if the callback succeeds. To make matters worse, this lock must be recursive, as there is nothing to prevent the callback from accessing the index and ID that occasioned the callback.

The correct solution here is to require that all such callbacks be multi-thread safe, and either be guaranteed to succeed, or that it be possible to roll back the effects of the callback. Unfortunately, this isn't an option at present, so the multi-thread safe version of H5I must simply accommodate the foibles of the non-multi-thread safe portions of the HDF5 library.

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

The other, H5CS (call stack) maintains a call stack for debugging purposes. From a brief examination of the code, it seems that if it 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.

6.2 Current Implementation of Multi-Thread Safe H5I

It was known from the start that retrofitting multi-thread safety on select HDF5 packages and integrating them into HDF5 library was going to involve a lot of cut and try. Thus, earlier versions of this document concentrated on describing the current state of H5I, challenges to be addressed, and possible solutions. No design work beyond a basic plan of attack was attempted due to the expectation that there would be constant adaption
to quirks in the existing code.

8 This is a bit of an overstatement. See the subsequent implementation discussion for a more accurate description of how this case is handled.

While this expectation was borne out by experience, a largely lock free, multi-thread safe implementation of H5I is now complete, and is passing the existing HDF5 regression tests, a new set of serial tests, and an initial set of multi-thread tests. Having reached this point, it is time to document the actual design and the reasons behind design decisions while they are still reasonably fresh in memory. Needless to say, the implementation will change as bugs are fixed and new quirks are encountered and adapted to. Thus, this should be a living document for some time to come.

The remainder of this section is organized into two subsections – one discussing the choice and implementation of the lock free indexing algorithm selected, and the other discussing the integration of this algorithm into H5I, and the modifications required to make H5I multi-thread safe. This latter topic becomes intertwined with integrating multi-thread safe H5I into the single thread portions of the HDF5 library.

6.2.1 The Lock Free Hash Table

The lock-free hash table that I selected to perform the basic indexing functions in H5I is described in section 13.3 of "The Art of Multiprocessor Programming" by Herlihy, Luchangco, Shawitt, and Spear.

In a nutshell, this algorithm computes a hash by inverting the bit order of the supplied ID, such that the least significant bit becomes the most significant bit, the second least significant bit becomes the second most significant bit, and so on. This done, the hash is left shifted one bit, with the new least significant bit set to one – this last step creates a separate namespace for sentinel nodes, of which more later. All entries in the hash table are stored in a lock-free singly linked list in increasing hash order.

Hash buckets are marked by sentinel nodes in the lock-free singly linked list whose hash value low order bits are zero. This ensures that no regular ID will hash to the hash value of a sentinel node, and thus the sentinel nodes can mark the beginnings of hash buckets without fear of collision. Pointers to the sentinel nodes are stored in a table. Given the hash value of any ID and the current maximum number of hash buckets (which must be a power of two), it is possible to compute the index in the sentinel pointer table for the associated hash bucket.

If the table contains a pointer to the associated sentinel, searches, insertion, and deletion proceed as per a regular lock free singly linked list[9] from that starting point. If the table doesn't contain a pointer to the associated sentinel, it must first be created and inserted – at which point operation proceeds as before.

The size of the table of sentinel node pointers is typically doubled whenever the ratio of hash table entries to sentinel pointer table size exceeds some user specified value[10]. Thus, in principle, the average number of nodes traversed in a successful search is limited to one half the ratio of hash table

9 For those unfamiliar with lock free singly linked lists, see section 10.5 of Herlihy, Luchangco, Shawitt, and Spear. Note that the authors call this data structure a "lock free unbounded queue".
10 But note that in the current implementation, the maximum table size is limited to 1024. This limitation will have to be removed prior to release of a production version.

entries to sentinel node pointer table size. While this needn't be the case with malevolently chosen IDs, ID's in H5I are allocated sequentially – and thus this average should be roughly correct.

From an API perspective, in addition to startup and shutdown calls, the lock free hash table supports the following operations that are used by H5I

- add
- find
- delete
- iteration (via get_first / get_next)

All of these operations are lock free, and serializable. Their use in H5I is discussed in detail in the next section[11].

This overview of the lock free hash table algorithm is highly simplified. As the remainder of this section presumes a reasonable understanding of this algorithm in particular, and of lock free algorithms and data structures in general, readers with any concerns on this point are advised to review Herlihy, Luchangco, Shawitt, and Spear, or some other similar text before proceeding.

The remainder of this section is directed at the particulars of the lock free hash table implementation, how the algorithm and data structure described in Herlihy, Luchangco, Shawitt, and Spear have been adapted for this use case, the state of the current implementation, and plans for the production implementation.

The header comment and type definition of the structure used to represent nodes in the lock free singly linked list is shown below:

```
/***********************************************************************************
* struct lfht_node_t
*
* Node in the lock free singly linked list that is used to store entries in the
* lock free hash table.  The fields of the node are discussed individually below:
*
* tag:        Unsigned integer set to LFHT_VALID_NODE whenever the node is either
*             in the LFSLL or the free list, and to LFHT_INVALID_NODE just before
*             it is discarded.
*
* next:       Atomic pointer to the next entry in the LFSLL, or NULL if
*             there is no next entry.  Note that due to the alignment guarantees of
*             malloc() & calloc(), the least significant few bits (at least three
*             in all cases investigated to date) will be zero.
*
*             This fact is used to allow atomic marking of the node for deletion.
*             If the low order bit of the next pointer is 1, the node is logically
```

11 Or will be in the final form of the document. At present, the next section restricts itself to data structures and general approach to synchronization problems. Detailed discussions of how operations are implemented will wait until H5I has a full regression test suite, and all issues exposed have been dealt with.

```
*           deleted from the LFSLL.  It will be physically deleted from the SLL
*           by a subsequent insert or delete call.  See section 9.8 of
*           "The Art of Multiprocessor Programming" by Herlihy, Luchangco,
*           Shavit, and Spear for further details.
*
* id:        ID associated with the contents of the node.  This field is
*           undefined if the node is a sentinel node
*
* hash:      For regular node, this is the hash value computed from the id.
*           For sentinel nodes, this is the smallest value that can map to
*           the associated hash table bucket -- see section 13.3.3 of
*           "The Art of Multiprocessor Programming" by Herlihy, Luchangco,
*           Shavit, and Spear for further details.
*
*           Note that duplicate hash codes cannot appear in the LFSLL, and that
*           nodes in the LFSLL appear in strictly increasing order.
*
* sentinel:   Boolean flag that is true if the node is a sentinel node, and
*           false otherwise.
*
* value:     Pointer to whatever structure is used to contain the value associated
*           with the id, or NULL if the node is a sentinel node.
*
*           This field is atomic, as we allow the client code to modify it in
*           an existing entry in the hash table.
*
************************************************************************************/

typedef struct lfht_node_t {

   unsigned int tag;
   struct lfht_node_t * _Atomic next;
   unsigned long long int id;
   unsigned long long int hash;
   bool sentinel;
   void * _Atomic value;

} lfht_node_t;
```

The first adaption to the C11 development environment appears in the next field, which is a pointer to the next entry in the lock free singly linked list (LFSLL). The algorithms for maintaining the LFSLL require that the flag indicating that an instance of lfht_node_t is logically deleted and the next pointer in that instance be modified in a single atomic operation.

In C11, this could be handled in either of two ways. The obvious approach would be to combine the deleted flag and the next pointer into an atomic structure. Depending on the host CPU, operations on this atomic structure could be handled as a true atomic operation, or the C11 run time would simulate an atomic operation by protecting the “atomic” structure with a mutex whenever it is either accessed or modified.

With the CPU of my development machine, the maximum size of true atomic structures is 64 bits. Thus for performance reasons I elected to go the second way. Specifically, I used the low order bit of the next pointer as the deleted flag. This works, as space allocated via malloc() and calloc() are guaranteed to be aligned with the largest scalar type supported by the host CPU – which means that the lower 3 bits of a pointer to dynamically allocated space will be zero given a 32 bit or larger CPU.

I subsequently learned that a 128 bit limit on the size of true atomic structures is common in more modern CPUs. Thus, if this optimization ever causes problems, there is an easy solution.

Before continuing, a few more observations about lfht_node_t are in order.

First, only the next and value fields are atomic. This works because all the other fields are constant whenever the instance is visible to more than one thread. If compilers ever start implementing cross function call optimizations, this will have to be revisited.

Finally, the value field is atomic, as the current implementation allow it to be modified during the lifetime of an entry in the lock free hash table. At present this capability is not used by H5I.

As is typical in lock free data structures, multiple threads may have pointers to lfht_node_t at the same time. This is a problem when it is time to physically delete a node from the LFSLL[12], as there is no guarantee that another thread will not attempt to access it after it has been physically deleted from the LFSLL. In languages that support garbage collection, this is simple – just drop the deleted node on the floor and let the garbage collection code return it to the heap.

Since C11 doesn't support garbage collection, another solution is required.

Conceptually, the idea is to place deleted nodes on a free list, and hold them there until all threads that might have a pointer to them have exited the lock free hash table – at which point they are eligible for either re-use or release to the heap. Note the hidden assumption that no pointer to an instance of lfht_node_t is retained by a thread after it exits a call to the lock free hash table.

To add the necessary infrastructure, instances of lfht_node_t are wrapped in instances lfht_fl_node_t. The type definitions of lfht_fl_node_t and lfht_flsprt_t are shown below.

```
/***********************************************************************************
* struct lfht_flsptr_t
*
* The lfht_flsptr_t combines a pointer to lfht_fl_node_t with a serial number in
* a 128 bit package.
*
* Unfortunately, this means that operations on atomic instances of this structure
* may or may not be truly atomic.  Instead, the C11 run time may maintain atomicity
* with a mutex.  While this may have performance implications, there should be no
* correctness implications.
*
* The combination of a pointer and a serial number is needed to address ABA
* bugs.
*
```

12 Recall that removing a node from the LFSLL is a two stage process. First is is marked as logically deleted, and then is is physically deleted in passing by a subsequent insert or delete operation

```
* ptr:        Pointer to an instance of flht_lf_node_t.
*
* sn:         Serial number that should be incremented by 1 each time a new
*           value is assigned to fl_ptr.
*
*************************************************************************************/

typedef struct lfht_flsptr_t {

   struct lfht_fl_node_t * ptr;
   unsigned long long int  sn;

} lfht_flsptr_t;


/*************************************************************************************
* struct lfht_fl_node_t
*
* Node in the free list for the lock free singly linked list on which entries
* in the lock free hash table are stored.  The fields of the node are discussed
* individually below:
*
* lfht_node:   Instance of lfht_node_t which is initialized and returned to the
*           lfht code by lfht_create_node.
*
*           If no node is available on the free list, an instance of
*           lfht_fl_node_t is allocated and initialized.  Its pointer is
*           then cast to a pointer to lfht_node_t, and returned to the
*           caller.
*
*           A node is available on the free list if the list contains more
*           that one entry, and the ref count on the first node in the list
*           is zero.  In this case, the first node is removed from the free
*           list, re-initialzed, its pointer cast to a pointer to lfht_node_t,
*           and returned to the caller.
*
* tag:         Atomic unsigned integer set to LFHT_FL_NODE_IN_USE whenever the
*           node is in the SLL, to LFHT_FL_NODE_ON_FL when the node is on the
*           free list, and to LFHT_FL_NODE_INVALID just before the instance of
*           lfht_fl_node_t is freed.
*
* ref_count:   If this instance of lfht_fl_node_t is at the tail of the free
*           list, the ref_count is incremented whenever a thread enters one
*           of the LFHT API calls, and decremented when the API call exits.
*
* sn:         Unique, sequential serial number assigned to each node when it
*           is placed on the free list.  Used for debugging.
*
* snext;       Atomic instance of struct lfht_flsptr_t, which contains a
*           pointer (ptr) to the next node on the free list for the lock
*           free singly linked list, and a serial number (sn) which must be
*           incremented each time a new value is assigned to snext.
*
*           The objective here is to prevent ABA bugs, which would
*           otherwise occasionally allow leakage of a node.
*
*************************************************************************************/

typedef struct lfht_fl_node_t {
    struct lfht_node_t lfht_node;
    _Atomic unsigned int tag;
    _Atomic unsigned int ref_count;
    _Atomic unsigned long long int sn;
    _Atomic struct lfht_flsptr_t snext;

} lfht_fl_node_t;
```

The first point of interest in lfht_fl_node_t is the snext field, which is an atomic instance of lfht_flsptr_t. This structure has two fields – a pointer to lfht_fl_node_t (ptr), and a serial number (sn) which must be incremented each time the pointer is modified.

This serialized pointer is used to prevent ABA bugs.

Such bugs occur when a test for a change in the value of a variable is used as a proxy to detect a change in a data structure. Thus, a thread may note that the relevant variable has value A, observe that the variable still has value A at a later date, and conclude that nothing has changed. However, this may be false if another thread has changed the value to B and then back to A in the interim. If a serial number is paired with the variable in an atomic structure, and incremented each time the variable is modified, a change in the variable can be detected, even if its value has been returned to that first observed.

ABA bugs can occur in the free list when entries are discarded, re-allocated, and discarded again in short order. This in turn can cause confusion with the free list tail pointer, resulting in memory leakage. The replacement of a simple next pointer with an instance of lfht_flsptr_t solves this problem.

The ref_count field is used to determine whether the node at the head of the free list is eligible for either re-allocation or release to the heap. The original idea was that each thread would increment the ref_count of the last entry in the free list on entry, keep a pointer to that entry, and then decrement its ref count on exit. Since no entry on the free list could be either re-allocated or released to the heap unless its ref_count was zero, and since all re-allocations or releases to the heap are made from the head of the free list, it follows that if a node is at the head of the free list with a zero ref_count, all threads that were in the lock free hash table at the time it was release have exited the lock free hash table.

While the basic idea is sound, it will not work as outlined above. The basic problem is that incrementing the ref_count of the node at the tail of the free list can't be an atomic operation. To see this, observe that one must first read the tail pointer (one atomic operation) and then increment the ref_count field on the indicated free list node (a second atomic operation). If the system is heavily loaded, it is possible for the target free list node to advance to the head of the free list and be re-allocated between these two operations.

Instead, each thread on entering the lock free hash table allocates a node, sets its ref_count to 1, releases it to the free list, and saves a pointer to the node. On exit, the thread uses this saved pointer to decrement the ref_count of the node. While this works, and is good enough for a working prototype, it must be revisited for the production version.

H5I must be able to create, initialize, clear, and delete indexes at any time during execution. To support this, the variables needed by the lock free hash table are packaged into a single structure, and there are functions to initialize it prior to use, and clear it prior to discard. The type definition and header comment for this structure (lfht_t) is shown below. Note that for brevity, the many fields used to collect statistics on the lock free hash table have been omitted. While they have great value for debugging, they don't add to this discussion.

```
/***************************************************************************************
 * struct lfht_t
 *
 * Root of a lock free hash table (LFHT).
 *
 * Entries in the hash table are stored in a lock free singly linked list (LFSLL).
 *
 * Each hash bucket has a sentinel node in the linked list that marks the
 * beginning of the bucket.  Pointers to the sentinel nodes are stored in
 * the index.
 *
 * Section 13.3.3 of "The Art of Multiprocessor Programming" by Herlihy,
 * Luchangco, Shavit, and Spear describes most of the details of the algorithm.
 * However, that discussion presumes implementation in a language with
 * garbage collection -- which simplifies matters greatly.
 *
 * The basic problem here is that we can't free a node that has been
 * removed from the LFSLL until we know that all references to it have been
 * discarded.  This is a problem, as an arbitrary number of threads may
 * have a pointer to a node at the point at which it is physically deleted
 * from the LFSLL.
 *
 * We solve this problem as follows:
 *
 * First, don't allow any node on the LFSLL to become visible outside of
 * the LFHT package.  As a result, we know that all pointers to a discarded
 * node have been discarded as well once all threads that were active in the
 * LFHT code at the point that the node was discarded have exited the LFHT
 * code.  We know this, as any such pointers must have been allocated on the
 * stack, and were therefore discarded when the associated threads left the
 * LFHT package.
 *
 * Second, maintain a free list of discarded nodes, and decorate each discarded
 * node with a reference count (see declarations of lfht_node_t and lfht_fl_node_t
 * above).
 *
 * Ideally, on entry to the LFHT package, each thread must increment the
 * reference count on the last node on the free list, and then decrement it
 * on exit. However, until I can find a way to make this operation atomic, this
 * is not workable, as the tail node may advance to the head of the free list
 * and be re-allocated in the time between the read of the pointer to the last
 * element on the free list, and the increment of the indicated ref_count.
 *
 * Instead, on entry to the LFHT package, each thread allocates a node, sets its
 * ref_count to 1, and releases it to the free list.  On exit, it decrements
 * the node's ref_count back to zero.  This has the same net effect, but is not as
 * efficient.  Needless to say, this issue should be revisited for the production
 * version.
 *
 * If we further require that nodes on the free list are only removed from
 * the head of the list (either for re-use or discard), and then only when their
 * reference counts are zero, we have that nodes are only released to the
 * heap or re-used if all threads that were active in LFHT package at the point
 * at which the node was place on the free list have since exited the LFHT
 * package.
 *
 * Between them, these two adaptions solve the problem of avoiding accesses to
 * nodes that have been returned to the heap.
 *
 * Finally, observe that the LFSLL code is simplified if it always contains two
 * sentinel nodes with (effectively) values negative and positive infinity --
 * thus avoiding operations that touch the ends of the list.
 *
 * In this context, the index sentinel node with hash value zero is created
 * at initialization time and serves as the node with value negative infinity.
 * However, since a sentinel node with hash positive infinity is not created
 * by the index, we add a sentinel node with hash LLONG_MAX at initialization
 * to serve this purpose.
 *
 * Note that the LFSLL used in the implementation of the LFHT is a modified
 * version of the lock free singly linked list discussed in chapter 9 of
 * "The Art of Multiprocessor Programming" by Herlihy, Luchangco, Shavit,
 * and Spear.
 *
 * The fields of lfht_t are discussed individually below.
 *
 *
 * tag:        Unsigned integer set to LFHT_VALID when the instance of lfht_t
 *           is initialized, and to LFHT_INVALID just before the memory for
 *           the instance of struct lfHT_t is discarded.
 *
 *
 * Lock Free Singly Linked List related fields:
 *
 * lfsll_root:  Atomic Pointer to the head of the SLL.  Other than during setup,
 *           this field will always point to the first sentinal node in the
 *           index, whose hash will be zero.
 *
 * lfsll_log_len:  Atomic integer used to maintain a count of the number of nodes
 *           in the SLL less the sentry nodes and the regular nodes that
 *           have been marked for deletion.
 *
 *           Note that due to the delay between the insertion or deletion
 *           of a node, and the update of the field, this count may be off
 *           for brief periods of time.
 *
 * lfsll_phys_len:  Atomic integer used to maintain a count of the actual number
 *           of nodes in the SLL.  This includes the sentry nodes, and any
 *           nodes that have been marked for deletion, but that have not
 *           been physically deleted.
 *
 *           Note that due to the delay between the insertion or deletion
 *           of a node, and the update of the field, this count may be off
 *           for brief periods of time.
 *
 *
```

```
* Free list related fields:
*
* fl_shead:    Atomic instance of struct lfht_flsptr_t, which contains a
*              pointer (ptr) to the head of the free list for the lock free
*              singly linked list, and a serial number (sn) which must be
*              incremented each time a new value is assigned to fl_shead.
*
*              The objective here is to prevent ABA bugs, which would
*              otherwise occasionally allow allocation of free list
*              nodes with positive ref counts.
*
* fl_stail:    Atomic instance of struct lfht_flsptr_t, which contains a
*              pointer (ptr) to the tail of the free list for the lock free
*              singly linked list, and a serial number (sn) which must be
*              incremented each time a new value is assigned to fl_stail.
*
*              The objective here is to prevent ABA bugs, which would
*              otherwise occasionally allow the tail of the free list to
*              get ahead of the head -- resulting in the increment of the
*              ref count on nodes that are no longer in the free list.
*
* fl_len:      Atomic integer used to maintain a count of the number of nodes
*              on the free list.  Note that due to the delay between free list
*              insertions and deletions, and the update of this field, this
*              count may be off for brief periods of time.
*
*              Further, since the free list must always contain at least one
*              entry.  When correct, fl_len will be one greater than the number
*              of nodes available on the free list.
*
* max_desired_fl_len: Integer field containing the desired maximum free list
*              length.  This is of necessity a soft limit as entries cannot
*              be removed from the head of the free list unless their
*              reference counts are zero.  Further, at most one entry is
*              removed from the head of the free list per call to
*              lfht_discard_node().
*
* next_sn:     Serial number to be assigned to the next node placed on the
*              free list.
*
*
* Hash Bucket Index:
*
* index_bits:  Number of index bits currently in use.
*
* max_index_bits:  Maximum value that index_bits is allowed to attain.  If
*              this field is set to zero, the lock free hash table becomes
*              a lock free singly linked list, as only one hash bucket is
*              permitted.
*
* index_masks: Array of unsigned long long containing the bit masks used to
*              compute the index into the hash bucket array from a hash code.
*
* buckets_defined: Convenience field.  This is simply 2 ** index_bits.
*              Needless to say, buckets_initialized must always be less than
*              or equal to buckets_initialized.
*
* buckets_initialized: Number of hash buckets that have been initialized --
*              that is, their sentinel nodes have been created, and inserted
*              into the LFSLL, and a pointer to the sentinel node has been
```

```
 *           copied into the bucket_idx.
 *
 * bucket_idx:  Array of pointers to lfht_node_t.  Each entry in the array is
 *           either NULL, or contains a pointer to the sentinel node marking
 *           the beginning of the hash bucket indicated by its index in the
 *           array.
 *
 * Statistics Fields:
 *
 * The following fields are used to record statistics on the operation of the
 * SLL for purposes of debugging and optimization.  All fields are atomic.
 *
 *  ****** Discussions of stats fields omitted for brevity ********
 *
 ***********************************************************************************/

#define LFHT__MAX_DESIRED_FL_LEN      256
#define LFHT__BASE_IDX_LEN           1024

typedef struct lfht_t
{
  unsigned int tag;


  /* LFSLL: */

  struct lfht_node_t * _Atomic lfsll_root;
  _Atomic long unsigned long int lfsll_log_len;
  _Atomic long unsigned long int lfsll_phys_len;


  /* Free List: */

  _Atomic struct lfht_flsptr_t fl_shead;
  _Atomic struct lfht_flsptr_t fl_stail;
  _Atomic long long int fl_len;
  int max_desired_fl_len;
  _Atomic unsigned long long int next_sn;


  /* hash bucket index */

  _Atomic int index_bits;
  int max_index_bits;
  unsigned long long int index_masks[LFHT__NUM_HASH_BITS + 1];
  _Atomic unsigned long long int buckets_defined;
  _Atomic unsigned long long int buckets_initialized;
  _Atomic (struct lfht_node_t *) bucket_idx[LFHT__BASE_IDX_LEN];


  /* statistics: */

  /**** Statistics fields omitted for brevity ****/

} lfht_t;
```

Aside from its existence, and the existence of functions to initialize it after it has been allocated, and clear it prior to its release, there are a few points worth noting about lfht_t.

First, note that the maximum number of hash buckets is currently hard coded to 1024. While this is adequate for a prototype, it does have performance implications in some cases, and should be reworked to allow the number of hash buckets to grow beyond 1024 if needed.

Second, note that as might be expected, the head and tail pointers of the free list are instances of lfht_flsptr_t. As discussed in the header comment, this prevents a number of ABA bugs.

Finally, in addition to the statistics fields which we have skipped over, lfht_t contains a number of fields that are included to facilitate debugging. For example, next_sn is used to assign a unique serial number to each node as it is inserted in the free list. Similarly, lfht_phys_len is used primarily for sanity checking and in memory leak detection.

A few points in closing this section.

At present, the lock free hash table is implemented as a collection of function calls. For the production version, it would be useful to re-work it into a collection of macros for both performance and portability reasons.

Second, as shall be seen in the next section, free lists similar to that maintained by the lock free hash table are needed frequently – and thus should be implemented as a collection of macros as well.

Third, while there is an extensive test suite for the lock free hash table, which runs without error on a variety of platforms, the code has not been subjected to any thread sanitizers. While this should be done, it will likely wait until it is near its final form.

6.2.2 Multi-Thread Safe H5I and its Integration with HDF5

While the multi-thread H5I code is largely complete, it has only been subjected to an initial smoke check. As more rigorous tests will trigger errors (for example ID not defined errors), further testing must be delayed until H5E has been updated to support multi-thread operations.

As it is inevitable that bugs will be encountered and that multi-thread safe H5I will be modified accordingly, a detailed discussion of multi-thread safe H5I is not appropriate at present. That said, multi-thread safe H5I must be documented while the relevant design decisions are fresh in mind.

To square this circle, this section discusses the data structure revisions needed to implement H5I multi-thread safe H5I, the general approaches used to address synchronization issues, and integration with single thread elements of the HDF5 library.

While the lock free hash table provides the core of a multi-thread safe H5I implementation, there are several issues remaining, which include:

- Lock free management of individual IDs in indexes.
- Lock free management of index creation, deletion, and lookup.

- Integration with the rest of HDF5 proper and VOL connectors.

While developing a lock free multi-thread safe version of H5I proper is quite do-able, integration with the single thread elements of HDF5 proper makes some locking inevitable. Callbacks that may or may not succeed and can't be rolled back complicate matters further, and raises the specter of deadlocks should a callback re-enter H5I – which at least one does.

Most of these issues center on individual entries in indexes, so we will start there.

For each entry in each index, H5I must maintain internal[13] and application reference counts, a pointer to the data associated with the ID, and flags indicating whether this entry is a future entry[14], or if it has been marked as deleted, but not yet removed from the index[15].

Further, the multi-thread implementation must execute the index specific free function to release the data associated with an entry when it is removed from the index. This function may or may not be multi-thread safe and may or may not succeed and can't be rolled back,

In addition, future entries have associated realize and discard callbacks. The realize callback is called whenever the associated future ID is searched for. If the realize callback fails, the search fails. If it succeeds, the realize callback returns the ID that is associated with the data needed to convert the future ID to a regular ID. The requires the following operations:

- Remove the ID returned by the realize callback from the index, saving a pointer to its associated data – call this pointer A.
- Call the discard callback to discard the data currently associated with the future ID.
- Set the void pointer associated with the future ID to A.
- Clear the is_future flag – converting the future ID to a regular ID.

This done, the lookup proceeds as normal.

Finally, HDF5 is old, complex, and poorly documented. Thus, at least until we have a working prototype, it seems prudent to make as few architectural changes as practical so as to reduce the chances of unexpected side effects.

H5I associates a structure with each ID it allocates. The single thread version of this structure is reproduced below:

13 Generally speaking, entries in the index are deleted when their internal reference counts drop to zero.

14 i.e. whether this entry is a placeholder for data that is not yet available.

15 This appears to be an adaption to users of the iteration calls which can't handle the contents of an index changing during the iteration. Obviously, this is a non-starter in the multi-thread safe case, so in the future, applications that have this limitation will have to either avoid multi-thread entirely, or enforce the necessary mutual exclusion themselves.

```
/* ID information structure used */
typedef struct H5I_id_info_t {
  hid_t       id;        /* ID for this info */
  unsigned    count;     /* Ref. count for this ID */
  unsigned    app_count; /* Ref. count of application visible IDs */
  const void *object;    /* Pointer associated with the ID */

  /* Future ID info */
  hbool_t                   is_future;  /* Whether this ID represents a */
                                        /* future object */
  H5I_future_realize_func_t realize_cb; /* 'realize' callback for future object */
  H5I_future_discard_func_t discard_cb; /* 'discard' callback for future object */

  /* Hash table ID fields */
  hbool_t        marked; /* Marked for deletion */
  UT_hash_handle hh;     /* Hash table handle (must be LAST) */
} H5I_id_info_t;
```

The hh field is specific to uthash and thus is not needed in the multi-thread safe version of this structure (called H5I_mt_id_info_t). All the remaining fields must be maintained in the multi-thread version.

Of these fields, id, realize_cb, and discard_cb either are or can be made constant for the life of the ID – and thus need not be atomic.

Of the remaining fields, count and app_count are the internal and application reference counts mentioned above. The object field is the void pointer to the data (if any) associated with the ID by the client. is_future is a flag indicating whether the ID is a future ID, and the marked flag indicates whether the ID has been logically deleted.

To avoid consistency issues between these fields, they and and two additional flags have been combined in a single structure, and made into the atomic field “k” (for kernel) in the new structure H5I_mt_id_info_t .

To maintain consistency, modifications to the kernel are done in an atomic read, modify local copy, and atomic write via compare_exchange_strong()[16] loop. This loop repeats until either the call to compare_exchange_strong() succeeds, or the point becomes moot. For example, just after the atomic read, we will always check the marked flag. If it is set, the ID has been deleted, so we must break out of the loop and report failure if appropriate.

The hidden assumption here is that modifications to the local copy of the kernel can be rolled back if we have to repeat the loop. This is true for H5I proper, but callbacks to functions that may fail and can’t be rolled back complicate matters.

16 compare_exchange_strong() will fail if the kernel has an unexpected value. Since we use the result of the previous atomic read as our expected value, the call will fail if the kernel has been modified since the atomic read.

The do_not_disturb flag was added to the kernel to address this issue. This flag is effectively a lock, as when set, it prevents any other thread from modifying the kernel until it is reset. To make this work, we test for the do_not_disturb flag just after checking the marked flag, and do a thread yield and return to the beginning of the loop[17] if it is set. This gives the setting thread time to complete its operation and reset the do_not_disturb flag.

When modification to the kernel involve operations that can't be rolled back (only callbacks at present), the procedure for modifying the kernel becomes more involved still:

1. Using an atomic read, obtain a local copy of the kernel. Call this copy A.
2. Test A to see if the marked flag is set. If it is, break out of the loop and report failure if appropriate
3. Test A to see if the do_not_disturb flag is set. It it is, thread yield and return to 1.
4. Make a second copy of A – call this copy B. Set the do_not_disturb flag in B.
5. Using a call to compare_exchange_strong() attempt to overwrite the kernel with B, using A as the expected value. If this fails, return to 1.
6. Make a copy of B – call this copy C. Reset the do_not_disturb flag in C.
7. Attempt to perform the operation that can't be rolled back. Depending on the success of the operation, alter C as appropriate.
8. Call compare_exchange_strong() to overwrite the kernel with C, using B as the expected value. This call must succeed as the do_not_disturb flag is set in the kernel.

The final new flag, have_global_mutex, is a temporary hack to deal with the case in which a callback accesses H5I and tries to operate on the same ID that triggered the callback in the first place. This flag is set at the same time as the do_not_disturb flag if the setting thread has the HDF5 global mutex. When set, it allows threads that have the global mutex to ignore the do_not_disturb flag. This situation arises in the HDF5 library, but fortunately, the callback doesn't modify the kernel in this case.

This hack must be revised to implement a more general recursive lock – probably by storing the thread ID when the do_not_disturb flag is set, and checking the stored thread ID and comparing it with the current thread ID when the do_not_disturb flag is encountered. It will also require logic to handle the case in which the callback modifies the kernel. However, since the have_global_mutex hack works for now, implementation of a cleaner solution has been delayed until multi-thread H5I is closer to its final form.

Since instances of H5I_mt_id_info_t can be simultaneously accessed by multiple threads, it follows that instances associated with deleted IDs must be maintained on a free list until no thread retains a pointer to them. While we will discuss this and another free list later, note that H5I_mt_id_info_t has fields needed to support such a free list.

17 i.e. execute the continue statement.

The header comment and definition of H5I_mt_id_info_t is reproduced below. Note that the header comment covers the above discussion of the kernel and its management in greater detail than sketched above.

```
/****************************************************************************************
 *
 * struct H5I_mt_id_info_t
 *
 * H5I_mt_id_info_t is a re-write of H5_id_info_t with modifications to facilitate
 * the multi-thread version of H5I.
 *
 * As such, most of the fields will be familiar from H5_id_info_t.
 *
 * Note that most of these are gathered together into a single, atomic sub-structure,
 * to allow atomic operations on the the id info.
 *
 * The remaining fields are either constant during the life of an instance of
 * H5I_mt_id_info_t, or exist to support the free list that a deleted instance of
 * H5I_mt_id_info_t must reside on until we are sure that no thread retains a pointer
 * to it.
 *
 * The fields of H5I_mt_id_info_t are discussed individually below.
 *
 * tag: unsigned int 32 set to H5I__ID_INFO when allocated, and to
 *      H5I__id_INFO_INVALID just before the instance of H5I_mt_id_info_t
 *      is deallocated.
 *
 * id:  ID associated with this instance of H5I_mt_id_info_t.  This is the id used to
 *      locate the instance in the lock free hash table.
 *
 *
 * k:   The non-MT version of H5I_mt_id_info_t has a number of variables that must be
 *      kept in synchronization.  The obvious way of doing this would be to protect
 *      them with a mutex.  However, it seems best to avoid locking to the extent
 *      possible so as to avoid lock ordering considerations.
 *
 *      This leads to the option of encapsulating the variables in a single atomic
 *      structure, the kernel for short.  In this case, the kernel must be read
 *      atomically, modified,  and written back atomically with a
 *      compare_exchange_strong().  In the event of failure in the
 *      compare_exchange_strong(), the procedure must be repeated
 *      until it is successful, or the point becomes moot – for example if the id info
 *      is marked as being deleted.
 *
 *      The hidden assumption here is that operations on the encapsulated variables
 *      can be rolled back if the compare_exchange_strong() fails.  This is clearly
 *      true with ref counts, and with overwrites of the pointer to the object
 *      associated with the id info.  If we further view an id as being functionally
 *      deleted once is it marked as deleted, in principle, this should be true of
 *      deletions as well, as we should be able to do the remainder of the cleanup
 *      at leisure.
 *
 *      By similar logic, this should include the discard of un-realized future IDs,
 *      as once the marked flag is set, no other will modify the kernel (see step 2 in
 *      the protocols below).
 *
 *      Unfortunately, the serial version of H5I doesn't work this way.  In
 *      particular, in H5I__mark_node(), if either the discard_cb (in the case of a
 *   future ID) or the free_func (in the case of a regular ID) fails, and the
 *   force flag is not set, the target ID is not marked for deletion, and the
 *   (possibly corrupt) buffer pointed to by the object field is left in the
 *   index.  This seems a questionable design choice, but until we have a working
 *   prototype, going with it seems to be the best option.  Obviously, it should
 *   be revisited once that point is reached.
 *
 *   Conversions from future to real IDs present a similar problem, as the
 *   conversion may fail, and cannot be rolled back.  Further, this operation may
 *   be attempted repeatedly until it succeeds.
 *
 *   To square this circle, we need a mechanism for serializing callbacks, and for
 *   ensuring that operations that can't be rolled back can't be interrupted.
 *
 *   We do this by adding two flags to the kernel -- the already defined
 *   marked flag, and the new do_not_disturb flag, and then proceeding by the
 *   appropriate protocol as given below.
 *
 *   In the cases where roll backs are possible, proceed as follows:
 *
 *    1) Load the kernel.
 *
 *    2) Check to see if the marked flag is set.  If so, issue an ID doesn't exist
 *       error and return.
 *
 *    3) Check to see if the do_not_disturb flag is set.  If so, do a thread yield
 *       or sleep, and return to 1) above.
 *
 *    4) Perform the desired operations (i.e ref count increment or decrement, or
 *       object pointer overwrite on the local copy of the kernel.  If the ref
 *       count drops to zero, set the marked flag on the local copy of the kernel.
 *
 *    5) Attempt to overwrite the global copy of the kernel with the local copy
 *       via a compare_exchange_strong().  If this succeeds, we are done.
 *       Otherwise roll back the operation, and return to 1.
 *
 *       Note no thread yield or sleep in this case, as this will typically be
 *       another thread that jumped in and modified the kernel.  If the
 *       do_not_disturb flag is set, we will hit it on the next pass.
 *
 *   Note that this thread yield or sleep and then re-try approach is also
 *   used in the lock free hash table to handle a very unlikely collisions without
 *   the use of locks in the lock free hash table.
 *
 *   For operations that can't be rolled back (i.e. realization or discard of
 *   future IDs, ID frees, etc), the above procedure is modified as follows:
 *
 *    1) Load the kernel.
 *
 *    2) Check to see if the marked flag is set.  If so, issue an ID doesn't exist
 *       error and return.
 *
 *    3) Check to see if the do_not_disturb flag is set.  If so, do a thread yield
 *       or sleep, and return to 1) above.
 *
 *    4) Check to see if the desired operation is still pending (i.e. if the
 *       operation is converting a future ID to a real ID, is the is_future flag
 *       still TRUE?). If it isn't, some other thread has already performed the
 *       operation so we can exit with success.
 *
```

```
 *      Otherwise:
 *
 *    5) Set the do_not_disturb flag in the local copy of the kernel, and attempt
 *       to overwrite the global copy of the kernel with the local copy via a
 *       compare_exchange_strong().
 *
 *       If this fails, do a thread yield or sleep, and return to 1.
 *
 *       If it succeeds, we know that we have exclusive access to the kernel until
 *       we reset the do_not_disturb flag on the global copy, as no new thread
 *       looking at the kernel will proceed beyond reading the flag, and the
 *       compare_exchange_strong() of any existing thread attempting to modify the
 *       kernel will fail -- sending it back to step 1.
 *
 *    6) Attempt to perform the desired operation.
 *
 *       If this fails, reset the do_not_disturb flag in the local copy of the
 *       kernel, overwrite the global copy of the kernel with the local copy via a
 *       compare_exchange_strong(), and report the failure of the operation if
 *       appropriate.
 *
 *       Observe that the call to compare_exchange_strong() must succeed, per the
 *       argument given in the final paragraph in 5 above.  Note that we use
 *       compare_exchange_strong() in this case only as a sanity check.  Assuming
 *       my analysis is correct, we could simply use atomic_store() on
 *       architectures where compare_exchange_strong() is not available.
 *       Unfortunately, the rest of the algorithm does depend on
 *       compare_exchange_strong(), so it will have to be re-worked for those
 *       architectures.
 *
 *       If the operation succeeds, update the kernel accordingly, reset the
 *       do_not_disturb flag in the local copy of the kernel, and overwrite the
 *       global copy of the  kernel with the local copy via a
 *       compare_exchange_strong().  As before this operation must succeed, and we
 *       are done.
 *
 *   An atomic instance of H5I_mt_id_info_kernel_t is used to instantiate the
 *   kernel mentioned above.  It maintains its fields as a single atomic object.
 *   As the size of this structure is too large for true atomic operations, C11
 *   maintains atomicity via mutexes.  This hurts performance, but since the
 *   objective is to avoid explicit locking (and thus lock ordering concerns) this
 *   is fine -- for now at least.
 *
 *   Note that if we combined all the booleans in a flags field,
 *   and reduced the size of the count and app_count integers, we could fit the
 *   H5I_mt_id_info_kernel_t into 128 bytes, allowing true atomic operation on
 *   many (most) more modern CPUs.  However, that is an optimization for another
 *   day, as is re-working the future ID feature into something more multi-thread
 *   friendly.
 *
 *   Since H5I_mt_id_info_kernel_t is only used either in H5I_mt_id_info_t, or to
 *   stage reads and writes of the kernel in that structure, its fields are
 *   discussed here.
 *
 *   k.count: Reference count on this ID.  This is typically the number of
 *       references to the ID elsewhere in the HDF5 library.  This ref count is
 *       used to prevent deletion of the id (and the associated instance of
 *       H5I_mt_id_info_t until all references have been dropped.
 *
 *   k.app_count: Application reference count on this ID.  This allows the
```

```
*          application to prevent deletion of this ID (under most circumstances)
*          until all its references to the ID have been dropped.
*
*      k.object: Pointer to void.  Points to the data (if any) associated with
*          this ID.
*
*      k.marked: Boolean flag indicating whether this instance of H5I_mt_id_info_t
*          has been marked for deletion.  Once set, this flag is never re-set, and
*          any ID for which this flag is set must be viewed as logically deleted,
*          even though the actual removal from the lock free has table and deletion
*          may occur later.
*
*      k.do_not_disturb: Boolean flag.  When set, a thread that needs to perform
*          an operation on the ID that can't be rolled back is in progress.  All
*          other threads must wait until this operation completes (see discussion
*          above).
*
*      k.is_future: Boolean flag indicating whether this ID represents a future ID.
*
*      k.have_global_mutex: Boolean flag that should be set to TRUE when
*          k.do_not_disturb is set to TRUE, and the setting thread has the HDF5
*          global mutex at the time.
*
*          This field is a temporary hack designed allow HDF5 callbacks to access
*          the index without deadlocking.  Thus, when the do_not_disturb flag is
*          detected, it can be ignored if the have_global_mutex flag is set and the
*          current thread has the global mutex.
*
*          It will almost certainly be replace with a thread ID stored in
*          H5I_mt_id_info_t proper, and set whenever k.do_not_disturb is set.  While
*          this will make k.do_not_disturb into a recursive lock, it will also
*          require additional logic to allow for the possibility that the kernel
*          has been modified while the k.do_not_disturb flag is set.
*
*      If we followed the single thread version of H5I exactly, the realize_cb and
*      discard_cb would have to be atomic since they are set to NULL when is_future
*      is set to FALSE.  However, that doesn't seem necessary, so the are non-atomic
*      fields in H5I_mt_id_info_t.  This should be OK, as the only time they are
*      modified is when the instance of H5I_mt_id_info_t is being initialized prior
*      to insertion into the index.  Since only one thread has access at that point,
*      leaving them as regular fields should work.  However, if compilers start
*      optimizing across function boundaries, this will have to be re-visited.
*
*      More generally, note that the above is a bit of a kluge to accommodate the
*      current implementation of future IDs, and more generally, to accommodate call
*      backs that can fail and/or can't be rolled back.
*
*      While we are probably stuck with the current callbacks for the native VOL
*      for the foreseeable future, new, more multi-thread friendly versions of the
*      H5I callbacks should be developed.
*
*      Finally, note that while we have technically managed to avoid locks, the
*      do_not_disturb flag is effectively a lock which will have to be made
*      recursive.  Its main virtue is its near total lack of overhead in cases
*      where locking is not required.  Hopefully, this will make up for its other
*      sins.
*
* realize_cb: 'realize' callback for future object.
*
* discard_cb: 'discard' callback for future object.
```

```
 *
 *
 * Fields supporting the H5I_mt_id_info_t free list:
 *
 * on_fl: Atomic boolean flag that is set to TRUE when the instance of
 *     H5I_mt_id_info_t is place on the id info free list, and to FALSE on initial
 *     allocation from the heap, or when the instance is allocated from the free
 *     list.
 *
 * re_allocable:  Atomic boolean flag that is set to FALSE on allocation from the
 *     heap or from the free list.  It is set to TRUE if the entry is on the free
 *     list and it is known that it is no longer on the lock free hash table, and
 *     no thread currently in H5I has a pointer to it.
 *
 * fl_snext: Atomic instance of H5I_mt_id_info_sptr_t used in the maintenance of the
 *     id info free list.  The structure contains both a pointer and a serial
 *     number, which facilitates the avoidance of ABA bugs when managing the free
 *     list.
 *
 ****************************************************************************************/

typedef struct H5I_mt_id_info_kernel_t {

  unsigned              count;     /* Ref. count for this ID */
  unsigned              app_count;  /* Ref. count of application visible IDs */
  const void           * object;    /* Pointer associated with the ID */

  hbool_t               marked;     /* Marked for deletion */
  hbool_t               do_not_disturb;
  hbool_t               is_future;  /* Whether this ID represents a future
                             object */
  hbool_t               have_global_mutex;

} H5I_mt_id_info_kernel_t;

typedef struct H5I_mt_id_info_t {

  uint32_t tag;

  hid_t id;

  _Atomic H5I_mt_id_info_kernel_t k;

  /* Future ID callbacks */
  H5I_future_realize_func_t realize_cb; /* 'realize' callback for future object */
  H5I_future_discard_func_t discard_cb; /* 'discard' callback for future object */

  _Atomic hbool_t on_fl;

  _Atomic hbool_t re_allocable;

  _Atomic H5I_mt_id_info_sptr_t fl_snext;

} H5I_mt_id_info_t;
```

While most of the multi-thread issues in H5I involve individual IDs in an index, H5I must also manage each individual index. The structures used for this in the single thread version are reproduced below:

```
typedef struct H5I_class_t {
   H5I_type_t type;     /* Class "value" for the type */
   unsigned   flags;     /* Class behavior flags */
   unsigned   reserved;  /* Number of reserved IDs for this type */
                     /* [A specific number of type entries may be
                      * reserved to enable "constant" values to be
                      * handed out which are valid IDs in the type,
                      * but which do not map to any data structures
                      * and are not allocated dynamically later.]
                      */
   H5I_free_t free_func; /* Free function for object's of this type */
} H5I_class_t;

typedef struct H5I_type_info_t {
   const H5I_class_t *cls;         /* Pointer to ID class */
   unsigned          init_count;   /* # of times this type has been initialized */
   uint64_t          id_count;     /* Current number of IDs held */
   uint64_t          nextid;       /* ID to use for the next object */
   H5I_id_info_t     *last_id_info; /* Info for most recent ID looked up */
   H5I_id_info_t     *hash_table;   /* Hash table pointer for this ID type */
} H5I_type_info_t;
```

Multi-thread H5I uses H5I_class_t without alteration, albeit with one newly defined flag. This flag is intended to mark indexes whose callbacks are “multi-thread safe”. The exact requirements for setting this flag are still to be determined – and will likely remain so for some time.

In contrast, H5I_type_info_t is replaced with the new structure H5I_mt_type_into_t in multi-thread H5I.

The cls field is constant, and thus remains unchanged. init_count, id_count, and next_id become atomic variables, but otherwise remain unchanged, and last_id_info becomes an atomic pointer to H5I_mt_id_info_t.

The hash_table field is specific to uthash, and thus is replace with two new fields – lfht, which is an instance of lfht_t – the root structure for a lock free hash table, and the atomic boolean flag lfht_cleared, which is set to TRUE when lfht has been cleared in preparation for discarding the index.

Like H5I_mt_id_info_t, H5I_mt_type_info_t can be accessed simultaneously by multiple threads. Thus, discarded instances of H5I_mt_type_info_t must be retained on a free list until it is certain that no thread retains a pointer to them. Also, as per H5I_mt_id_info_t, the pointers used in this list must be combined with a serial number in an atomic structure, with the serial number being incremented each time the pointer is modified.

While this may change as I get deeper into testing multi-thread index creation and deletion mixed with operations on the indexes in question, so far it hasn’t seemed necessary to combine any of the fields of H5I_mt_type_info_t into an atomic substructure as was done in H5I_mt_id_info_t.

The type definitions of H5I_mt_type_info_sptr_t and H5I_mt_type_info_t are reproduced below, along with their header comments.

```
/****************************************************************************************
 *
 * struct H5I_mt_type_info_sptr_t
 *
 * H5I_mt_type_info_sptr_t combines a pointer to H5I_mt_type_info_t with a serial
 * number that must be incremented each time the value of the pointer is changed.
 *
 * When it appears in either H5I_mt_type_info_t or H5I_mt_t, it should do so
 * as an atomic object.  Its purpose is to avoid ABA bugs.
 *
 * ptr: pointer to an instance of H5I_mt_type_info_t, or NULL if undefined.
 *
 * sn:  uint64_t that is initialized to 0, and must be incremented by 1 each time the
 *      ptr field is modified.
 *
 ****************************************************************************************/

typedef struct H5I_mt_type_info_t H5I_mt_type_info_t; /* forward declaration */

typedef struct H5I_mt_type_info_sptr_t {

  H5I_mt_type_info_t * ptr;
  uint64_t           sn;

} H5I_mt_type_info_sptr_t;

/****************************************************************************************
 *
 * struct H5I_mt_type_info_t
 *
 * H5I_mt_type_info_t is a re-write of H5_type_info_t with modifications to facilitate
 * the multi-thread version of H5I.
 *
 * As such, most of the fields will be familiar from H5_type_info_t.
 *
 * tag: unsigned int 32 set to H5I__TYPE_INFO when allocated, and to
 *      H5I__TYPE_INFO_INVALID just before the instance of H5I_mt_id_info_t is
 *      deallocated.
 *
 * cls: Pointer to the ID class.
 *
 * init_count: Number of times this type has been initialized less the number of times
 *      its reference count has been decremented.
 *
 * id_count: Current number of IDs in the type.
 *
 * next_id: ID to be allocated to the next object inserted into the index.
 *
 * last_id_info: Pointer to the instance of H5I_mt_id_info_t associated with the last
 *      ID accessed in the index.  Note that it is possible for this pointer to be
 *      NULL, or for it to point (briefly) to and instance of H5I_mt_id_info_t that
 *      has been marked for deletion.
 *
 * lfht_cleared: Boolean flag that is set to TRUE when the lock free hash table
 *      associated with the index is cleared in preparation for deletion.
 *
 * lfht: The instance of lfht_t that forms the root of the lock free hash table in
 *      which all objects in the index are stored, and which supports the look up of
 *      the instance of H5I_mt_id_info_t associated with any given ID.
 *
 *     Note that the lock free hash table may contain pointers to instances of
 *     H5I_mt_id_info_t that have been marked for deletion.  Such entries and their
 *     associated IDs have been logically deleted from the index, even their
 *     associated instances of H5I_mt_id_info_t remain in the lock free hash table.
 *
 *
 * Fields supporting the H5I_mt_type_info_t free list:
 *
 * on_fl: Atomic boolean flag that is set to TRUE when the instance of
 *     H5I-mt_type_info_t is place on the type info free list, and to FALSE on
 *     initial allocation from the heap, or when the instance is allocated from
 *     the free list.
 *
 * re_allocable: aAtomic boolean flag that is set to FALSE on allocation from the heap
 *     or from the free list.  It is set to TRUE if the entry is on the free list and
 *     it is known that it is no longer on the lock free hash table, and no thread
 *     currently in H5I has a pointer to it.
 *
 * fl_snext: Atomic instance of H5I_mt_type_info_sptr_t used in the maintenance of the
 *     type info free list.  The structure contains both a pointer and a serial
 *     number, which facilitates the avoidance of ABA bugs when managing the free
 *     list.
 *
 ***************************************************************************************/

typedef struct H5I_mt_type_info_t {
  uint32_t                    tag;
  const H5I_class_t          * cls;        /* Pointer to ID class */
  _Atomic unsigned             init_count;   /* # of times this type has been
                              * initialized */
  _Atomic uint64_t            id_count;     /* Current number of IDs held */
  _Atomic uint64_t            nextid;       /* ID to use for the next object */
  H5I_mt_id_info_t * _Atomic      last_id_info; /* Info for most recent ID looked
                              * up */
  _Atomic hbool_t             lfht_cleared; /* TRUE iff the lock free hash table
                              * has been cleared in prep for
                              * deletion */
  lfht_t                   lfht;       /* lock free hash table for this
                              * ID type */
  _Atomic hbool_t             on_fl;
  _Atomic hbool_t             re_allocable;
  _Atomic H5I_mt_type_info_sptr_t fl_snext;
} H5I_type_info_t;
```

Finally, H5I must create and discard indexes as directed – and thus requires the supporting data structures.

Note that for historical reasons, indexes are referred to as “types” in the H5I code. While confusing from the point of view of this paper, it is probably due to the fact that each index is used to store, index, and assign IDs to different types of data – and from that perspective it makes sense. However, as this paper is concerned with the internals of H5I, we will continue to call indexes indexes.

In the single thread version of H5I, the data structures needed to manage multiple indexes (AKA types) are implemented as a small collection of global and static variables – reproduced below:

```
H5_DLLVAR H5I_type_info_t *H5I_type_info_array_g[H5I_MAX_NUM_TYPES];

/* Variable to keep track of the number of types allocated.  Its value is the
* next type ID to be handed out, so it is always one greater than the number
* of types.
* Starts at 1 instead of 0 because it makes trace output look nicer.  If more
* types (or IDs within a type) are needed, adjust TYPE_BITS in H5Ipkg.h
* and/or increase size of hid_t
*/
H5_DLLVAR int H5I_next_type_g;

/* Whether deletes are actually marks (for mark-and-sweep) */
static hbool_t H5I_marking_s = FALSE;
```

H5I_type_info_array_g[] is used to store pointers to the defined indexes and their associated data stored in instances of H5I_type_info_t. Null entries indicate index (AKA type) IDs that are currently undefined.

If H5I_next_type_g global is less than H5I_MAX_NUM_TYPES, it is used to store the next index (AKA type) ID that is available for allocation. Failing that, the single thread code scans H5I_type_info_array_g[] looking for a NULL cell. If one is found, the associated index (AKA type) ID is allocated, and that cell in the array is set to point to the instance of H5I_type_info_t that contains the data associated with the new index (AKA type). The index creation fails if no vacant location is found on the H5I_type_info_array_g[].

When set, H5I_marking_s forces IDs that would otherwise be removed from any index, to be marked for deletion, but not actually deleted. The presumption is that a later sweeps of the indexes will physically remove the marked entries.

This general architecture is retained in multi-thread H5I, albeit with significant adjustments.

The first of these is to collect all fields associated with the management of indexes (AKA types) into a single structure (H5I_mt_t), a single instance of which is declared as a global (H5I_mt_g).

The array of pointers to the root structures of the defined indexes (AKA types) is retained, although its name has been changed to H5I_mt_g.type_info_array[], each of whose elements is an atomic pointer to H5I_mt_id_info_t. As before, this array is indexed by the IDs assigned to the various indexes (AKA types). However, a companion array of atomic booleans (H5I_mt_g.type_info_allocation_table) has been added to track which index (AKA type) IDs have been allocated. This allows the assignment of index IDs and the creation of indexes to be managed as two separate, atomic operations.

The H5I_next_type_g global is retained as H5I_mt_g.next_type, and functions as before save that H5I_mt_g.type_info_allocation_table[] is scanned for available IDs, instead of H5I_mt_g.type_info_array[] in the event that H5I_mt_g.next_type is greater than or equal to H5I_MAX_NUM_TYPES.

The marking boolean is also retained as H5I_mt_g.marking, and functions as before. Note, however, that the semantics of the marked flag in individual indexes has changed, as entries in indexes are now logically deleted as soon as their marked flags are set. (Note: double check this)

In addition to these legacy fields, H5I_mt_t has many other fields, that fall into two general categories.

The first of these is management of the free lists for discarded instances of H5I_mt_id_info_t and H5I_mt_type_info_t. These lists function much as the free list in the lock free hash table, but note that facilities for determining when no thread retains a pointer to an entry are presently a sketch, and have not been implemented. As errors in this element of free list implementation are frequently indistinguishable from errors elsewhere in lock free multi-thread code, I have found it prudent to simply retain all entries on the free list pending shutdown until associated code is fully debugged.

The second major category is statistics. While statistics collection is of immense value in debugging complex code in general and multi-thread code in particular, it is of no great interest in the current context. Thus, the statistics fields have been omitted in the following reproduction of the definition of H5I_mt_t and its associated header comment.

```
/****************************************************************************************
 *
 * struct H5I_mt_t
 *
 * A single, global instance of H5I_mt_t is used to collect all H5I global variables
 * that:
 *
 * 1) are existing globals that must be made into atomic variables in the MT context,
 *    or
 *
 * 2) are new globals needed for the MT case, or
 *
 * 3) are needed to maintain statistics on the multi-thread version of H5I.
 *
 * The single instance is declared as a global, and initialized in H5I_init().
 *
 * Fields are discussed individually below.
 *
 *
 * Pre-existing Globals:
 *
 * type_info_array: Array of atomic pointers to H5I_mt_type_info_t of length
 *      H5I_MAX_NUM_TYPES used to map type IDs to the associated instances of
 *      H5I_mt_type_info_t,
 *
 *      All elements of the type_info_array are initialized to NULL, and are
 *      set back to NULL when the associated instance of H5I_mt_type_info_t
 *      is discarded.
 *
 * type_info_allocation_table:  Array of atomic booleans used to track which
 *      entries in the type_info_array are allocated.
 *
 *      In the single thread version of H5I, type IDs can be re-used.  I had
 *      originally planed to simplify the H5I code by dis-allowing this in the
 *     multi-thread H5I code, however this breaks the serial tests.  While it
 *     is doubtful that this has any practical significance, this is not
 *     something to be changed lightly -- particularly in the initial prototype.
 *
 *     To make things more complicated, type IDs are allocated in two different
 *     ways.  Index types used by the HDF5 library proper are allocated statically
 *     in the header files, while index types created by users are allocated
 *     dynamically in H5Iregister_type().  In the single thread code, this is
 *     done via one of two methods:
 *
 *     First, type IDs are allocated sequentially using the H5I_next_type_g global
 *     to maintain the next ID to be allocated.  This method is used until
 *     the value of H5I_next_type_g is equal to H5I_MAX_NUM_TYPES.  When this
 *     method is exhausted, the sequential code does a linear scan on the
 *     H5I_type_info_array_g, and uses the index of the first NULL entry
 *     encountered as the next type ID to allocate.
 *
 *     The H5I_next_type_g has been retained in the multi-thread case as
 *     the atomic next_type field in this structure, and that the first method
 *     of assigning type IDs has been retained otherwise unchanged.
 *
 *     The second method of assigning type IDs by scanning the type_info_array is
 *     not directly applicable to the multi-thread case as the ID is allocated
 *     well before the associated entry in the type_info_array is set to point to
 *     a new instance of H5I_mt_type_info_t.  While this could be finessed with a
 *     special value indicating that the entry in the type_info_array was allocated
 *     but invalid, an allocation table seemed to offer a cleaner solution.
 *
 *     Thus the booleans in the type_info_allocation_table are set to TRUE when
 *     a type ID is allocated, and back to FALSE when it is de-allocated.  In
 *     contrast, the pointers in the type_info_array are NULL if the associated
 *     ID is undefined, and point to the associated instance of H5I_mt_type_info_t
 *     if the index is defined.  This lets us treat index type ID allocation and
 *     index type definition as two, separate atomic operations.
 *
 * next_type: Atomic integer.  If its value is less than H5I_MAX_NUM_TYPES, it
 *     contains the next type ID to be assigned.  Otherwise, it is a flag indicating
 *     that the type_info_allocation_table must be scanned for an un-used type ID.
 *
 * marking: Functionally, the marking field replaces the H5I_marking_g boolean
 *     in the single thread version of H5I.  It is an integer instead of a
 *     boolean to allow for multiple threads setting and re-setting it.
 *
 *     In the serial version, when H5I_marking_g is set, when entries are deleted,
 *     they are first marked (i.e. info_ptr->k.marked is set to TRUE), an then
 *     deleted from the index at a later point in time.
 *
 *     The argument for this is that callbacks may delete entries on the index
 *     and thus cause data structure corruption.
 *
 *     However, in the multi-thread case, entries can be deleted whenever, so
 *     the point seems moot -- we simply need to adjust to that fact.
 *
 *     However, to minimize code turbulence, this field has been retained for now,
 *     and is used as per the single thread version.
 *
 *
 * New Globals:
 *
 * active_threads: Atomic integer used to track the number of threads currently
 *      active in H5I.
 *
 * id_info_fl_shead: Atomic instance of struct H5I_mt_id_info_sptr_t, which contains
 *      a pointer (ptr) to the head of the id info free list, and a serial number (sn)
 *      which must be incremented each time a new value is assigned to
 *      id_info_fl_shead.ptr.
 *
 *      The objective here is to prevent ABA bugs.
 *
 *      Note that once initialized, the id info free list will always contain at least
 *      one entry, and is logically empty if:
 *
 *              id_info_fl_shead.ptr ==  id_info_fl_stail.ptr != NULL.
 *
 * id_info_fl_stail: Atomic instance of struct H5I_mt_id_info_sptr_t, which contains
 *      a pointer (ptr) to the tail of the id info free list, and a serial number (sn)
 *      which must be incremented each time a new value is assigned to
 *      id_info_fl_stail.
 *
 *      The objective here is to prevent ABA bugs.
 *
 * id_info_fl_len: Atomic unsigned integer used to maintain a count of the number of
 *      nodes on the id info free list.  Note that due to the delay between free list
 *      insertions and deletions, and the update of this field, this count may be off
 *      for brief periods of time.
 *
 *      Recall that the free list must always contain at least one entry.  Thus, when
 *      correct, fl_len will be one greater than the number of entries on the free
 *      list.
 *
 * max_desired_id_info_fl_len: Unsigned integer field containing the desired maximum
 *      id info free list length.  This is of necessity a soft limit as entries cannot
 *      be removed from the head of the free list unless their re-allocable fields are
 *      TRUE.
 *
 *
 * type_info_fl_shead: Atomic instance of struct H5I_mt_type_info_sptr_t, which
 *      contains a pointer (ptr) to the head of the type info free list, and a serial
 *      number (sn) which must be incremented each time a new value is assigned to
 *      type_info_fl_shead.
 *
 *      The objective here is to prevent ABA bugs.
 *
 *      Note that once initialized, the type info free list will always contain at
 *      least one entry, and is logically empty if:
 *
 *             type_info_fl_shead.ptr ==  type_info_fl_stail.ptr != NULL.
 *
 * type_info_fl_stail: Atomic instance of struct H5I_mt_type_info_sptr_t, which
 *      contains a pointer (ptr) to the tail of the type info free list, and a serial
 *      number  (sn) which must be incremented each time a new value is assigned to
 *      type_info_fl_stail.ptr.
 *
 *      The objective here is to prevent ABA bugs.
 *
 * type_info_fl_len: Atomic unsigned integer used to maintain a count of the number of
 *      nodes on the type info free list.  Note that due to the delay between free
 *      list insertions and deletins, and the update of this field, this count may be
 *      off for brief periods of time.
 *
 *     Recall that the free list must always contain at least one entry.  Thus, when
 *     correct, type_info_fl_len will be one greater than the number of entries on
 *     the free list.
 *
 * max_desired_type_info_fl_len: Unsigned integer field containing the desired maximum
 *     type info free list length.  This is of necessity a soft limit as entries
 *     cannot be removed from the head of the free list unless their re-allocable
 *     fields are TRUE.
 *
 * Statistics:
 *
 * dump_stats_on_shutdown: Boolean flag that controls display of statistics in
 *     H5I_term_package().  When set to TRUE, stats are displayed when shutdown is
 *     complete, and just before stats are reset.
 *
 *
 * ======= Statistics fields omitted for brevity ==========
 *
 ****************************************************************************************/
typedef struct H5I_mt_t {

 /* Pre-existing Globals: */
 _Atomic (H5I_mt_type_info_t *) type_info_array[H5I_MAX_NUM_TYPES];
 _Atomic hbool_t type_info_allocation_table[H5I_MAX_NUM_TYPES];
 _Atomic int next_type;
 _Atomic int marking;

 /* New Globals: */
 _Atomic uint32_t active_threads;

 _Atomic H5I_mt_id_info_sptr_t   id_info_fl_shead;
 _Atomic H5I_mt_id_info_sptr_t   id_info_fl_stail;
 _Atomic uint64_t               id_info_fl_len;
 uint64_t                      max_desired_id_info_fl_len;

  _Atomic H5I_mt_type_info_sptr_t type_info_fl_shead;
  _Atomic H5I_mt_type_info_sptr_t type_info_fl_stail;
  _Atomic uint64_t               type_info_fl_len;
  uint64_t                      max_desired_type_info_fl_len;

  /* Statistics: /

  _Atomic hbool_t dump_stats_on_shutdown;

  /***** stats fields omitted for brevity *****/

} H5I_mt_t;
```

While most of the general approach to integrating multi-thread H5I with the single thread sections of HDF5 have already been discussed, there is one last detail that remains to be discussed.

H5I is called extensively from the single thread sections of HDF5. Since these sections of HDF5 must be protected by the global mutex, it follows that the global mutex will be held while these threads are in H5I. Thus, to avoid extra activity on the global mutex, we check whether the current thread holds the global mutex just before callback invocations, and only wrap the callback in the global mutex if

the mutex is not already
held.

7 Appendix 1 – H5I public API calls

After some type and macro definitions, this appendix contains a list of all the public H5I API calls, along with call trees, relevant structure definitions, and descriptions of their processing with particular emphasis on multi-thread safety issues. Note that this data was derived by inspection, and thus some errors and/or oversights should be expected.

The list of public API calls was taken from H5Ipublic.h and H5Idev.h. All the public API calls in this file are decorated with Doxygen code to generate user level documentation on public API calls. I have included this code as it may be a useful addition to my own documentation.

Finally, I have not investigated H5Iget_file_id() and H5Iget_name() beyond construction of an initial call tree, as these functions contain direct calls to H5VL, and H5F, and to other packages farther down the call tree. As the focus of the current effort is H5I, I am bypassing these calls for now.

```
/**
 * Library type values.
 * \internal Library type values.  Start with `1' instead of `0' because it
 *           makes the tracing output look better when hid_t values are large
 *           numbers. Change the TYPE_BITS in H5I.c if the MAXID gets larger
 *           than 32 (an assertion will fail otherwise).
 *
 *           When adding types here, add a section to the 'misc19' test in
 *           test/tmisc.c to verify that the H5I{inc|dec|get}_ref() routines
 *           work correctly with it. \endinternal
 */
//! <!-- [H5I_type_t_snip] -->
typedef enum H5I_type_t {
    H5I_UNINIT = (-2),  /**< uninitialized type                         */
    H5I_BADID  = (-1),  /**< invalid Type                               */
    H5I_FILE   = 1,     /**< type ID for File objects                   */
    H5I_GROUP,          /**< type ID for Group objects                  */
    H5I_DATATYPE,       /**< type ID for Datatype objects               */
    H5I_DATASPACE,      /**< type ID for Dataspace objects              */
    H5I_DATASET,        /**< type ID for Dataset objects                */
    H5I_MAP,            /**< type ID for Map objects                    */
    H5I_ATTR,           /**< type ID for Attribute objects              */
    H5I_VFL,            /**< type ID for virtual file layer             */
    H5I_VOL,            /**< type ID for virtual object layer           */
    H5I_GENPROP_CLS,    /**< type ID for generic property list classes */
    H5I_GENPROP_LST,    /**< type ID for generic property lists         */
    H5I_ERROR_CLASS,    /**< type ID for error classes                  */
    H5I_ERROR_MSG,      /**< type ID for error messages                 */
    H5I_ERROR_STACK,    /**< type ID for error stacks                   */
    H5I_SPACE_SEL_ITER, /**< type ID for dataspace selection iterator  */
    H5I_EVENTSET,       /**< type ID for event sets                     */
    H5I_NTYPES          /**< number of library types, MUST BE LAST!    */
} H5I_type_t;

H5Iprivate.h:#define H5I_IS_LIB_TYPE(type) (type > 0 && type < H5I_NTYPES)
```

```
/****************************/
/* Package Private Typedefs */
/****************************/

/* ID information structure used */
typedef struct H5I_id_info_t {
   hid_t          id;               /* ID for this info */
   unsigned    count;           /* Ref. count for this ID */
   unsigned    app_count;  /* Ref. count of application visible IDs */
   const void *object;        /* Pointer associated with the ID */

   /* Future ID info */
   hbool_t                             is_future;      /* Whether this ID represents a
future object */
   H5I_future_realize_func_t realize_cb;    /* 'realize' callback for future object */
   H5I_future_discard_func_t discard_cb;  /* 'discard' callback for future object */

   /* Hash table ID fields */
   hbool_t               marked; /* Marked for deletion */
   UT_hash_handle hh;          /* Hash table handle (must be LAST) */
} H5I_id_info_t;

/* Type information structure used */
typedef struct H5I_type_info_t {
   const H5I_class_t *cls;               /* Pointer to ID class */
   unsigned               init_count;   /* # of times this type has been initialized */
   uint64_t                 id_count;     /* Current number of IDs held */
   uint64_t                 nextid;          /* ID to use for the next object */
   H5I_id_info_t          *last_id_info; /* Info for most recent ID looked up */
   H5I_id_info_t          *hash_table;  /* Hash table pointer for this ID type */
} H5I_type_info_t;

/****************************/
/* Library Private Typedefs */
/****************************/

typedef struct H5I_class_t {
   H5I_type_t type;       /* Class "value" for the type */
   unsigned   flags;       /* Class behavior flags */
   unsigned   reserved; /* Number of reserved IDs for this type */
                                     /* [A specific number of type entries may be
                                      * reserved to enable "constant" values to be
                                      * handed out which are valid IDs in the type,
                                      * but which do not map to any data structures
                                      * and are not allocated dynamically later.]
                                      */
   H5I_free_t free_func; /* Free function for object's of this type */
} H5I_class_t;
```

H5I Public API calls:

```
/**
 * \ingroup H5IUD
 *
 * \brief Registers an object under a type and returns an ID for it
 *
 * \param[in] type The identifier of the type of the new ID
 * \param[in] object Pointer to object for which a new ID is created
 *
 * \return \hid_t{object}
 *
 * \details H5Iregister() creates and returns a new ID for an object.
 *
 * \details The \p type parameter is the identifier for the ID type to which
 *          this new ID will belong. This identifier must have been created by
 *          a call to H5Iregister_type().
 *
 * \details The \p object parameter is a pointer to the memory which the new ID
 *          will be a reference to. This pointer will be stored by the library
 *          and returned via a call to H5Iobject_verify().
 *
 */
H5_DLL hid_t H5Iregister(H5I_type_t type, const void *object);

H5Iregister()
 +-H5I__register()
```

In a nutshell:

H5Iregister() inserts the supplied void pointer in the index of the indicated type, and returns an ID that can be used to access this void pointer at later date.

In more detail:

The function tests to see if the supplied type is a library type (i.e. one used internally). It fails if it is. Otherwise it calls H5I__register() with the app_ref. realize_cb, and discard_cb parameters set to TRUE, NULL, and NULL respectively.

H5I__register() performs some sanity checks and flags errors if they fail (interaction with H5E). Assuming success, it allocates a new instance of H5I_id_info_t via a call to H5FL_CALLOC(), constructs a new ID via a call to the H5I_MAKE macro, loads the new instance of H5I_id_info_t, and inserts it into the hash table associated with the type via a call to the HASH_ADD() macro.

Note that the object provided for insertion in the index is simply stored by reference (i.e. only a pointer is saved). Since the caller may retain a pointer to this object, the index has no control over access to it. Thus maintaining mutual exclusion on this object to avoid corruption must be the caller's responsibility.

Finally, before returning, H5I__register() makes note of the most recent ID referenced of this type.

H5I__register() returns the new ID, which is returned to the caller by H5Iregister().

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate the supplied type, and to look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index.

Read / write access to *type_info for purposes of:

- Allocating the next ID (type_info->nextid)
- Incrementing the number of IDs in the index (type_info->id_count)
- Setting the last id touched (type_info->last_id_info)
- Inserting the instance of H5I_id_info_t associated with the new id into the hash table associated with the type (type_info->hash_table)

Use of H5FL_CALLOC() to allocate the instance of H5I_id_info_t used to store the new ID and its void *.

```
/**
 * \ingroup H5IUD
 *
 * \brief Returns the object referenced by an ID
 *
 * \param[in] id ID to be dereferenced
 * \param[in] type The identifier type
 *
 *
 * \return Pointer to the object referenced by \p id on success, NULL on failure.
 *
 * \details H5Iobject_verify() returns a pointer to the memory referenced by id
 *          after verifying that \p id is of type \p type. This function is
 *          analogous to dereferencing a pointer in C with type checking.
 *
 * \note H5Iobject_verify() does not change the ID it is called on in any way
 *       (as opposed to H5Iremove_verify(), which removes the ID from its
 *       type’s hash table).
 *
 * \see H5Iregister()
 *
 */
H5_DLL void *H5Iobject_verify(hid_t id, H5I_type_t type);

H5Iobject_verify()
 +-H5I_object_verify()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nutshell:

H5Iobject_verify() looks up the supplied ID & type pair, and, absent errors, returns the void pointer that was associated with this ID and type in a previous H5Iregister() or H5Iregister_future() call.

In more detail:

H5Iobject_verify() first tests to see if the supplied type either out of range, or is a library type (i.e. one used internally by the HDF5 library). It fails if either of these conditions are true. Otherwise, it calls H5I_object_verify() with the supplied parameters, and returns that function's return value.

H5I_object_verify() verifies that the type is in range -- that is that it is greater than zero and less than the H5I_next_type_g global, and that the supplied type matches the supplied ID via H5I_TYPE() macro.

If all sanity checks pass, H5I_object_verify() calls H5I__find_id(). Absent errors, H5I__find_id() returns a pointer to the instance of H5I_id_info_t associated with the ID in the target index -- call this pointer info. H5I_object_verify() returns info->object, which is the void pointer associated with the ID in a previous register call.

H5I__find_id() obtains the type associated with the supplied ID via the H5I_type() macro, validates that it is in range, and looks up the pointer to the associated instance of H5I_type_info_t in the H5I_type_info_array_g global array, and stores this pointer in type_info.

If type_info is not NULL, and type_info->init_count is positive, it looks up the target id, checking type_info->last_id_info first, and using the HASH_FIND() macro if that fails. The result of this search is stored in the local variable id_info, and in type_info->last_id_info – both of which will be NULL if the target id is not found.

If the search is successful (i.e., id_info is not NULL), H5I__find_id() tests to see if the index entry is a "future" entry (i.e., if id_info->is_future). This is an uncommon case, used only (to my knowledge) by the asynchronous VOL.

If it is, H5I__find_id() calls the user provided realize callback (id_info->realize_cb)((void *)id_info->object, &actual_id) which was provided in H5Iregister_future() (see below).

While I have not located any documentation specifying the behavior of the user supplied realize_cb function, from context it seems that it is supposed to find the actual ID associated with the future_id if it exists, and return it in *actual_id. If (id_info->realize_cb)() fails in this for whatever reason, H5I__find_id() returns NULL.

Assuming the actual ID is found, H5I__find_id() makes note of the future object, and calls H5I__remove_common() both to delete the actual ID from the index, and to return the object associated with the actual ID. The function then sets the object associated with the future ID equal to the object associated with the actual id (i.e. sets id_info->object = actual_object;), and calls the discard_cb to discard the object previously associated with the future ID. Finally, it converts the "future" index entry to an actual entry by setting:

```
id_info->is_future  = FALSE;
id_info->realize_cb = NULL;
id_info->discard_cb = NULL;
```

Finally, id_info is returned to the caller

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate the supplied type, and to look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index.

Read / write access to *type_info for purposes of:

- Reading type_info→init_count.
- Obtain a pointer (id_info) to the instance of H5I_id_info_t associated with the target ID.
- Setting the last ID touched (type_info→last_id_info).

Read access to *id_info for purposes of reading:

- id_info→is_future and
- id_info→object.

If id_info→is_future is TRUE, matters become much more involved from a thread safety perspective as H5I__find_id() attempts to convert the future id into the actual id. Assuming that it is successful, this involves the following additional accesses to data that is accessible to other threads:

Read of the H5I_marking_g global.

R/W access to *type_info to lookup of the instance of H5I_id_info_t associated with the actual id, and (if H5I_marking_g is FALSE) removal of the associated instance of H5I_id_info_t from the hash table.

R/W access to the instance of H5I_id_info_t associated with the actual id. In particular read of the object field, and (if H5I_marking_g is TRUE), set of the marked field. Finally, (if H5I_marking_g is FALSE), free of the instance via H5FL_FREE().

R/W access to the instance of H5I_id_info_t associated with the supplied ID (*id_info). In particular:

- execution of the realize_cb() to obtain the actual id,
- execution of the discard_cp to discard id_info→object,
- set id_info→object equal to the object field of the actual id
- set id_info→realize_cb = NULL
- set id_info→discard_cb = NULL
- set id_info→is_future = FALSE

Note that these accesses are spread across a number of functions – which complicates matters further.

```
/**
 * \ingroup H5IUD
 *
 * \brief Removes an ID from its type
 *
 * \param[in] id The ID to be removed from its type
 * \param[in] type The identifier type
 *
 *
 * \return Returns a pointer to the memory referred to by \p id on success,
 *         NULL on failure.
 *
 * \details H5Iremove_verify() first ensures that \p id belongs to \p type.
 *          If so, it removes \p id from its type and returns the pointer
 *          to the memory it referred to. This pointer is the same pointer that
 *          was placed in storage by H5Iregister(). If id does not belong to
 *          \p type, then NULL is returned.
 *
 *          The \p id parameter is the ID which is to be removed from its type.
 *
 *          The \p type parameter is the identifier for the ID type which \p id
 *          is supposed to belong to. This identifier must have been created by
 *          a call to H5Iregister_type().
 *
 * \note This function does NOT deallocate the memory that \p id refers to.
 *       The pointer returned by H5Iregister() must be deallocated by the user
 *       to avoid memory leaks.
 *
 */
H5_DLL void *H5Iremove_verify(hid_t id, H5I_type_t type);

H5Iremove_verify()
 +-H5I__remove_verify()
    +-H5I_remove()
       +-H5I__remove_common()
```

In a nutshell:

Delete the index entry associated with the supplied id, and return the void pointer that was supplied at registration. If the H5I_marking_g global is TRUE, just mark the entry for deletion without actually deleting it at this time.

In more detail:

H5Iremove_verify() verifies that the supplied type is not a HDF5 library internal type. If it is, the function flags an error and returns NULL.

Assuming that this test passes, the function calls H5I__remove_verify().
and returns whatever value that function returns.

H5I__remove_verify() verifies that the supplied type and id match (via H5I_TYPE() -- returning NULL without flagging an error if they do not. If they do, it calls H5I_remove() and returns whatever value that function returns.

H5I_remove() looks up the type embedded in the supplied ID, verifies that it is valid, and looks up the associated type info. It then calls H5I__remove_common() passing a pointer to this type info and the supplied ID as parameters. The function saves the value returned by H5I__remove_common() and returns this value

Using the supplied type info, H5I__remove_common() looks up the supplied ID in the hash table associated with the supplied type_info using HASH_FIND(). If the associated instance of H5I_id_info_t is not found, the function flags an error and returns NULL.

If the associated instance of H5I_id_info_t is found, H5I__remove_common() tests the H5I_marking_g global.

If H5I_marking_g is FALSE, the instance of H5I_id_info_t is removed from the type specific hash table via HASH_DELETE(). If H5I_marking_g is TRUE, the marked field of the instance of H5I_id_info_t is set to TRUE.

In either case, if target id was the last id of this type accessed, type_info->last_id_info is set to NULL (thread safety), and the return value of the function is set equal to the void pointer that was provided when the id was registered.

If H5I_marking_g is FALSE, the target instance of H5I_id_info_t is freed via H5FL_FREE().

The number of IDs of the target type is decremented, and the function returns.

Multi-Thread safety concerns:

Read access to the H5I_next_type_g and H5I_type_info_array_g global variables to validate and look up a pointer (type_info) to the instance of H5I_type_info_t associated with the target index. Also read access to the H5I_marking_g global to determine how to implement the removal.

Read / write access to *type_info for purposes of:

- Reading type_info→init_count.
- Obtain a pointer (id_info) to the instance of H5I_id_info_t associated with the target ID.
- Reading the last id touched (type_info→last_id_info) and setting it to NULL if it equals id_info.
- Decrementing type_info->id_count

Read / write access to *id_info for purposes of reading:

- setting id_info→marked = TRUE if H5I_marking_g is TRUE
- reading id_info→object.

Freeing *id_info via H5FL_FREE().

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves the type of an object
 *
 * \obj_id{id}
 *
 * \return Returns the object type if successful; otherwise #H5I_BADID.
 *
 * \details H5Iget_type() retrieves the type of the object identified by
 *          \p id. If no valid type can be determined or the identifier submitted is
 *          invalid, the function returns #H5I_BADID.
 *
 *          This function is of particular use in determining the type of
 *          object closing function (H5Dclose(), H5Gclose(), etc.) to call
 *          after a call to H5Rdereference().
 *
 * \note Note that this function returns only the type of object that \p id
 *       would identify if it were valid; it does not determine whether \p id
 *       is valid identifier. Validity can be determined with a call to
 *       H5Iis_valid().
 *
 */
H5_DLL H5I_type_t H5Iget_type(hid_t id);

H5Iget_type()
 +-H5I_get_type()
 +-H5I_object()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nutshell:

Return the type to which the supplied id belongs.

In more detail:

The type of an ID is encoded in the ID.

Thus H5Iget_type() calls H5I_get_type() which invokes the H5I_TYPE() macro to extract the type from the supplied ID, and returns this value to H5Iget_type().

H5Iget_type() then verifies that the type is valid, and that the supplied id is valid. If both tests pass, the type is returned to the caller. If not, an error is flagged and H5I_BADID is returned.

The bad type test accesses the H5I_next_type_g global.

The test for the validity of the supplied ID calls H5I_object(), which attempts to look up the ID, and return the void pointer that was supplied in the register call that created the ID. It does this via a call to H5I__find_id() which returns a pointer (info) to the instance of H5I_id_info_t associated with the id, or NULL if the search fails. If the search succeeds, the function returns info→object.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id(), the only multi-thread safety concern in H5Iget_type() is read access to the H5I_next_type_g global.

In contrast, H5I__find_id() has significant multi-thread safety issues – particularly if the target id is a future id. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
**
 * \ingroup H5I
 *
 * \brief Retrieves an identifier for the file containing the specified object
 *
 * \obj_id{id}
 *
 * \return \hid_t{file}
 *
 * \details H5Iget_file_id() returns the identifier of the file associated with
 *          the object referenced by \p id.
 *
 * \note Note that the HDF5 library permits an application to close a file
 *       while objects within the file remain open. If the file containing the
 *       object \p id is still open, H5Iget_file_id() will retrieve the
 *       existing file identifier. If there is no existing file identifier for
 *       the file, i.e., the file has been closed, H5Iget_file_id() will reopen
 *       the file and return a new file identifier. In either case, the file
 *       identifier must eventually be released using H5Fclose().
 *
 * \since 1.6.3
 *
 */
H5_DLL hid_t H5Iget_file_id(hid_t id);

H5Iget_file_id()
 +-H5VL_vol_object()
 |  +-H5I_get_type()
 |  +-H5I_object()
 |  |  +-H5I__find_id()
 |  |     +-(id_info->realize_cb)()
 |  |     +-H5I__remove_common()
 |  |     +-(id_info->discard_db)()
 |  +-H5T_get_named_type()
 +-H5F_get_file_id()
    +-H5VL_object_get()
    +-H5I__find_id()
    |  +-(id_info->realize_cb)()
    |  +-H5I__remove_common()
    |  +-(id_info->discard_db)()
    +-H5VL_set_vol_wrapper()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |  |      ||
    |  |    H5CX__get_context()
    |  |     +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |  |     +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    |  +-((vol_obj->connector->cls->wrap_cls.get_wrap_ctx)(vol_obj->data, &obj_wrap_ctx)
    |  +-H5VL_conn_inc_rc()
    |  +-H5CX_set_vol_wrap_ctx()
    |     +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |         ||
    |       H5CX__get_context()
    |        +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |        +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    +-H5VL_wrap_register()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +- ... see above
    |  +-H5T_already_vol_managed()
    |  +-H5VL__wrap_obj()
    |  |  +-H5CX_get_vol_wrap_ctx()
    |  |  |  +- ... see above
    |  |  +-H5VL_wrap_object()
    |  |    +-(connector->wrap_cls.wrap_object)(obj, obj_type, wrap_ctx)
    |  +-H5VL_register_using_vol_id()
    |     +-H5VL_new_connector()
    |     |  +-H5I_object_verify()
    |     |  |  +-H5I__find_id()
    |     |  |     +-(id_info->realize_cb)()
    |     |  |     +-H5I__remove_common()
    |     |  |     +-(id_info->discard_db)()
    |     |  +-H5I_inc_ref()
    |     |  |  +-H5I__find_id()2Y
    |     |  |     +- ... see above
    |     |  +-H5I_dec_ref()
    |     |     +-H5I__dec_ref()
    |     |        +-H5I__find_id()2Y
    |     |        |  +- ... see above
    |     |        +-(type_info->cls->free_func)((void *)info->object, request)
    |     |        +-H5I__remove_common()
    |     +-H5VL_register()
    |     |  +-H5VL__new_vol_obj()
    |     |  |  +-H5VL__wrap_obj()
    |     |  |     +- ... see above
    |     |  +-H5I_register()
    |     |     +-H5I__register()
    |     +-H5VL_conn_dec_rc()
    |        +-H5I_dec_ref()
    |           +- ... see above
    +-H5I_inc_ref()
    |  +-H5I_dec_ref()
    |     +- ... see above
    +-H5VL_reset_vol_wrapper()
       +-H5CX_get_vol_wrap_ctx()
       |  +- ... see above
       +-H5VL__free_vol_wrapper()
       |  +-(*vol_wrap_ctx->connector->cls->wrap_cls.free_wrap_ctx)(vol_wrap_ctx-
>obj_wrap_ctx)
       |  +-H5VL_conn_dec_rc()
       |     +- ... see above
       +-H5CX_set_vol_wrap_ctx()
          +- ... see above
```

Skipped for now due to calls to H5VL, H5CX, H5F, and H5T.

Return to this call after reviewing H5VL and H5CX.

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves a name of an object based on the object identifier
 *
 * \obj_id{id}
 * \param[out] name A buffer for thename associated with the identifier
 * \param[in] size The size of the \p name buffer; usually the size of
 *                 the name in bytes plus 1 for a NULL terminator
 *
 * \return ssize_t
 *
 * \details H5Iget_name() retrieves a name for the object identified by \p id.
 *
 * \details Up to size characters of the name are returned in \p name;
 *          additional characters, if any, are not returned to the user
 *          application.
 *
 *          If the length of the name, which determines the required value of
 *          \p size, is unknown, a preliminary H5Iget_name() call can be made.
 *          The return value of this call will be the size in bytes of the
 *          object name. That value, plus 1 for a NULL terminator, is then
 *          assigned to size for a second H5Iget_name() call, which will
 *          retrieve the actual name.
 *
 *          If the object identified by \p id is an attribute, as determined
 *          via H5Iget_type(), H5Iget_name() retrieves the name of the object
 *          to which that attribute is attached. To retrieve the name of the
 *          attribute itself, use H5Aget_name().
 *
 *          If there is no name associated with the object identifier or if the
 *          name is NULL, H5Iget_name() returns 0 (zero).
 *
 * \note Note that an object in an HDF5 file may have multiple paths if there
 *       are multiple links pointing to it. This function may return any one of
 *       these paths. When possible, H5Iget_name() returns the path with which
 *       the object was opened.
 *
 * \since 1.6.0
 *
 */
H5_DLL ssize_t H5Iget_name(hid_t id, char *name /*out*/, size_t size);

H5Iget_name()
 +-H5VL_vol_object()
 |  +-H5I_get_type()
 |  +-H5I_object()
 |  |  +-H5I__find_id()
 |  |     +-(id_info->realize_cb)()
 |  |     +-H5I__remove_common()
 |  |     +-(id_info->discard_db)()
 |  +-H5T_get_named_type()
 +-H5I_get_type()
 +-H5VL_object_get()
    +-H5VL_set_vol_wrapper()
    |  +-H5CX_get_vol_wrap_ctx()
    |  |  +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |  |     ||
    |  |    H5CX__get_context()
    |  |     +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |  |     +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    |  +-((vol_obj->connector->cls->wrap_cls.get_wrap_ctx)(vol_obj->data, &obj_wrap_ctx)
    |  +-H5VL_conn_inc_rc()
    |  +-H5CX_set_vol_wrap_ctx()
    |     +-H5CX_get_my_context() macro -- resolves to H5CX__get_context() in MT
    |        ||
    |       H5CX__get_context()
    |        +-H5TS_get_thread_local_value() -- pthread_getspecific() in most cases
    |        +-H5TS_set_thread_local_value() -- pthread_setspecific() in most cases
    +-H5VL__object_get()
    |  +-(cls->object_cls.get)(obj, loc_params, args, dxpl_id, req)
    +-H5VL_reset_vol_wrapper()
       +-H5CX_get_vol_wrap_ctx()
       |  +- ... see above
       +-H5VL__free_vol_wrapper()
       |  +-(*vol_wrap_ctx->connector->cls->wrap_cls.free_wrap_ctx)(vol_wrap_ctx-
>obj_wrap_ctx)
       |  +-H5VL_conn_dec_rc()
       |     +- ... see above
       +-H5CX_set_vol_wrap_ctx()
          +- ... see above
```

Skipped for now due to calls to H5VL, H5CX, and H5T.

Return to this call after reviewing H5VL and H5CX.

```
/**
 * \ingroup H5I
 *
 * \brief Increments the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative reference count of the object ID after
 *         incrementing it if successful; otherwise a negative value is
 *         returned.
 *
 * \details H5Iinc_ref() increments the reference count of the object
 *          identified by \p id.
 *
 *          The reference count for an object ID is attached to the information
 *          about an object in memory and has no relation to the number of
 *          links to an object on disk.
 *
 *          The reference count for a newly created object will be 1. Reference
 *          counts for objects may be explicitly modified with this function or
 *          with H5Idec_ref(). When an object ID's reference count reaches
 *          zero, the object will be closed. Calling an object ID's \c close
 *          function decrements the reference count for the ID which normally
 *          closes the object, but if the reference count for the ID has been
 *          incremented with this function, the object will only be closed when
 *          the reference count reaches zero with further calls to H5Idec_ref()
 *          or the object ID's \c close function.
 *
 *          If the object ID was created by a collective parallel call (such as
 *          H5Dcreate(), H5Gopen(), etc.), the reference count should be
 *          modified by all the processes which have copies of the ID.
 *          Generally this means that group, dataset, attribute, file and named
 *          datatype IDs should be modified by all the processes and that all
 *          other types of IDs are safe to modify by individual processes.
 *
 *          This function is of particular value when an application is
 *          maintaining multiple copies of an object ID. The object ID can be
 *          incremented when a copy is made. Each copy of the ID can then be
 *          safely closed or decremented and the HDF5 object will be closed
 *          when the reference count for that that object drops to zero.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Iinc_ref(hid_t id);

H5Iinc_ref()
 +-H5I_inc_ref()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nut shell:

Find the target id, and increment both its regular and applications reference counts. Return the new value of the application reference count.

If the target ID is a future ID, in passing, attempt to convert it to a real ID. Note that this attempt may cause the function to fail.

In greater detail:

H5Iinc_ref() calls H5I_inc_ref() with the supplied ID, and the app_ref parameter set to TRUE.

H5I_inc_ref() calls H5I__find_id() to obtain a pointer to the instance of H5I_id_info_t associated with the ID. Assuming that this is successful, the function increments both the regular an application reference counts, and returns the new value of the application reference count.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id(), the only multi-thread safety concern in H5Iget_type() is read / write access to the count and app_count fields of the target instance of H5I_id_info_t .

In contrast, H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
 * \ingroup H5I
 *
 * \brief Decrements the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative reference count of the object ID after
 *         decrementing it, if successful; otherwise a negative value is
 *         returned.
 *
 * \details H5Idec_ref() decrements the reference count of the object
 *          identified by \p id.
 *
 *          The reference count for an object ID is attached to the information
 *          about an object in memory and has no relation to the number of
 *          links to an object on disk.
 *
 *          The reference count for a newly created object will be 1. Reference
 *          counts for objects may be explicitly modified with this function or
 *          with H5Iinc_ref(). When an object identifier’s reference count
 *          reaches zero, the object will be closed. Calling an object
 *          identifier’s \c close function decrements the reference count for
 *          the identifier which normally closes the object, but if the
 *           reference count for the identifier has been incremented with
 *           H5Iinc_ref(), the object will only be closed when the reference
 *           count reaches zero with further calls to this function or the
 *           object identifier’s \c close function.
 *
 *           If the object ID was created by a collective parallel call (such as
 *           H5Dcreate(), H5Gopen(), etc.), the reference count should be
 *           modified by all the processes which have copies of the ID.
 *           Generally this means that group, dataset, attribute, file and named
 *           datatype IDs should be modified by all the processes and that all
 *           other types of IDs are safe to modify by individual processes.
 *
 *           This function is of particular value when an application is
 *           maintaining multiple copies of an object ID. The object ID can be
 *           incremented when a copy is made. Each copy of the ID can then be
 *           safely closed or decremented and the HDF5 object will be closed
 *           when the reference count for that that object drops to zero.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Idec_ref(hid_t id);

H5Idec_ref()
 +-H5I_dec_app_ref()
    +-H5I__dec_app_ref()
       +-H5I__dec_ref()
       |  +-H5I__find_id()
       |  |  +-(id_info->realize_cb)()
       |  |  +-H5I__remove_common()
       |  |  +-(id_info->discard_db)()
       |  +-(type_info->cls->free_func)((void *)info->object, request)
       |  +-H5I__remove_common()
       +-H5I__find_id()
          +-(id_info->realize_cb)()
          +-H5I__remove_common()
          +-(id_info->discard_db)()
```

In a nutshell:

Decrement both the regular and application reference counts on the target id. If the regular reference count drops to zero, delete the target instance from the index.

If the target ID is a future ID, in passing, attempt to convert it to a real ID. Note that this attempt may cause the function to fail.

In more detail:

After some sanity checks, H5Idec_ref() calls H5I_dec_app_ref(), and returns its return value.

H5I_dec_app_ref() is basically a pass through. It preforms some sanity checks, and the calls H5I__dec_app_ref(id, H5_REQUEST_NULL), and returns whatever value H5I__dec_app_ref() returns.

H5I__dec_app_ref() calls H5I__dec_ref() to decrement the regular ref count on the target. If H5I__dec_ref() returns a positive value (indicating that the regular reference count has not been decremented to zero), the function calls H5I__find_id() to obtain a pointer to the instance of

H5I_id_info_t associated with the ID. This in hand, the function decrements the application reference count. The function returns either the value return by H5I__dec_ref() (if it is non-positive), or the application reference count after it has been decremented.

H5I__dec_ref() first calls H5I__find_id() to obtain a pointer (info) to the instance of H5I_id_info_t associated with the target index entry.

If info→count is greater that one, it decrements that value, and returns it to the caller.

If info→count is one, it accesses the H5I_type_info_array_g global to look up the pointer to the instance of H5I_type_info_t associated with the target, calls type_info→free_func() (if it exists) to free info→object, calls H5I__remove_common() to remove *info from the index, and returns 0.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

H5I__remove_common() looks up the target ID in the index to obtain a pointer (info) to the associated instance of H5I_id_info_t.

If the H5I_marking_g global is FALSE, it removes *info from the index, and frees it via a call to H5FL_FREE().

If the H5I_marking_g global is TRUE, it sets info→marked = TRUE.

In either case, it decrements type_info→id_count, and returns info→object.

Multi-Thread safety concerns:

Leaving aside H5I__find_id() and H5I_remove_common() (which is called by H5I__find_id(), and thus included in its discussion), the multi-thread safety concerns in H5Idec_ref() are:

- Read access to H5I_type_info_array_t to obtain a pointer (type_info) to the instance of H5I_type_info_t associated with the index containing the target id.
- Execution of type_info→free_func()
- Decrement of info→count and info→app_count.

H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
 * \ingroup H5I
 *
 * \brief Retrieves the reference count for an object
 *
 * \obj_id{id}
 *
 * \return Returns a non-negative current reference count of the object
 *         identifier if successful; otherwise a negative value is returned.
 *
 * \details H5Iget_ref() retrieves the reference count of the object identified
 *          by \p id.
 *
 *          The reference count for an object identifier is attached to the
 *          information about an object in memory and has no relation to the
 *          number of links to an object on disk.
 *
 *          The function H5Iis_valid() is used to determine whether a specific
 *          object identifier is valid.
 *
 * \since 1.6.2
 *
 */
H5_DLL int H5Iget_ref(hid_t id);

H5Iget_ref()
 +-H5I_get_ref()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

In a nut shell:

Lookup the supplied id. If it exists, return its application reference count. If the call fails for any reason, return -1.

In more detail:

After some sanity checks, H5Iget_ref() calls H5I_get_ref() with the app_ref parameter equal to TRUE. It returns whatever value H5I_get_ref() returns.

H5I_get_ref() calls H5I__find_id() to look up the target index entry and return a pointer (info) to the instance of H5I_id_info_t associated with the id. If it is successful, H5I_get_ref() returns the current value of either the regular or the application reference count as directed by the app_ref parameter.

H5I__find_id() is discussed in some detail in the section on H5Iobject_verify() above -- thus no need to repeat that discussion here.

Multi-Thread safety concerns:

Leaving aside H5I__find_id() , the multi-thread safety concerns in H5Iget_ref() are read access to either info→count or info→app_count, depending on the value of the app_ref parameter passed to H5I_get_ref().

H5I__find_id() has significant multi-thread safety issues – particularly if the target ID is a future ID. See the discussion of thread safety for H5Iobject_verify() above for a full discussion.

```
/**
```

```
 * \ingroup H5IUD
 *
 * \brief Creates and returns a new ID type
 *
 * \param[in] hash_size Minimum hash table size (in entries) used to store IDs
 *                      for the new type
 * \param[in] reserved Number of reserved IDs for the new type
 * \param[in] free_func Function used to deallocate space for a single ID
 *
 * \return Returns the type identifier on success, negative on failure.
 *
 * \details H5Iregister_type() allocates space for a new ID type and returns an
 *          identifier for it.
 *
 *          The \p hash_size parameter indicates the minimum size of the hash
 *          table used to store IDs in the new type.
 *
 *          The \p reserved parameter indicates the number of IDs in this new
 *          type to be reserved. Reserved IDs are valid IDs which are not
 *          associated with any storage within the library.
 *
 *          The \p free_func parameter is a function pointer to a function
 *          which returns an herr_t and accepts a \c void*. The purpose of this
 *          function is to deallocate memory for a single ID. It will be called
 *          by H5Iclear_type() and H5Idestroy_type() on each ID. This function
 *          is NOT called by H5Iremove_verify(). The \c void* will be the same
 *          pointer which was passed in to the H5Iregister() function. The \p
 *          free_func function should return 0 on success and -1 on failure.
 *
 */
H5_DLL H5I_type_t H5Iregister_type(size_t hash_size, unsigned reserved, H5I_free_t
free_func);

H5Iregister_type()
 +-H5I_register_type()
```

In a nutshell:

Create a new type of index as specified, and return its ID. On failure, return a negative value.

In more detail:

H5Iregister_type() first attempts to allocate an ID for the new type.

If H5I_next_type_g is less than H5I_MAX_NUM_TYPES, it sets new_type = H5I_next_type_g and then increments H5I_next_type_g.

If this approach fails, it scans the global H5I_type_info_array_g array skipping library defined types looking for a NULL entry. If it finds one, it sets new_type equal to the index of the NULL entry.

If this second approach fails, the function fails.

Assuming an ID can be allocated for the new type, H5Iregister_type() allocates an instance of H5I_class_t via a call to H5MM_calloc(), initializes it with the data provided and the new ID, and then calls H5I_register_type() to perform the actual registration.

H5I_register_type() allows multiple registrations of a given type – of which more in the discussion of multi-thread safety concerns.

After some initial sanity checks, H5I_register_type() reads the new id from the supplied instance of H5I_class_t (cls->type), and then examines the global H5I_type_info_array_g array at that index (H5I_type_info_array_g[cls->type]). If that index contains NULL, it allocates a new instance H5I_type_info_t (via H5MM_calloc), sets type_info to point to it, and sets H5I_type_info_array_g[cls->type] = type_info.

If H5I_type_info_array_g[cls->type] is not NULL, the function sets type_info = H5I_type_info_array_g[cls->type].

The function tests type_info->init_count. If it is zero it initializes *type_info.

Finally, it increments type_info->init_count and returns.

Multi-Thread safety concerns:

In the absence of any access control on the H5I_next_type_g and H5I_type_info_array_g global variables, the current algorithm for allocating type IDs has a number of race conditions which appear to make it possible for a given ID to be allocated more than once – not to mention the possibility that other threads that require only read access to these variables will see them in an inconsistent state.

In addition, H5I_register_type() acceptance of multiple registrations of a given type present potential race conditions potentially resulting in data corruption unless calls for a given type are somehow serialized.

```
/**
 * \ingroup H5IUD
 *
 * \brief Deletes all identifiers of the given type
 *
 * \param[in] type Identifier of identifier type which is to be cleared of identifiers
 * \param[in] force Whether or not to force deletion of all identifiers
 *
 * \return \herr_t
 *
 * \details H5Iclear_type() deletes all identifiers of the type identified by
 *          the argument \p type.
 *
 *          The identifier type's free function is first called on all of these
 *          identifiers to free their memory, then they are removed from the
 *          type.
 *
 *          If the \p force flag is set to false, only those identifiers whose
 *          reference counts are equal to 1 will be deleted, and all other
 *          identifiers will be entirely unchanged. If the force flag is true,
 *          all identifiers of this type will be deleted.
 *
 */
H5_DLL herr_t H5Iclear_type(H5I_type_t type, hbool_t force);

/* User data for H5I__clear_type_cb */
typedef struct {
    H5I_type_info_t *type_info; /* Pointer to the type's info to be cleared */
    hbool_t          force;     /* Whether to always remove the ID */
    hbool_t          app_ref;   /* Whether this is an appl. ref. call */
} H5I_clear_type_ud_t;

H5Iclear_type()
 +-H5I_clear_type()
    +-H5I__mark_node()
       +-H5I__mark_node()
```

In a nutshell:

Delete all IDs of the target type with ref count 1 (i.e. not in current use) from the index. If the force flag is set, delete all IDs of the target type regardless of ref count.

In more detail:

H5Iclear_type() verifies that the supplied type is not a library type, and then calls H5I_clear_type() with the supplied type and force parameters, and with the app_ref parameter set to TRUE. It returns whatever value H5I_clear_type() returns.

Using the H5I_next_type_g and H5I_type_info_array_g global variables, H5I_clear_type() validates the supplied type, and looks up a pointer (udata.type_info) to the instance of H5I_type_info_t associated with the target type, and loads it into its user data (instance of H5I_clear_type_ud_t – see above) along with the force and app_ref parameters.

It then sets the H5I_marking_g global to TRUE, and uses the uthash HASH_ITER macro to set up a for loop to scan through all the entries in the hash table associated with the target id type -- calling H5I__mark_node() on each such entry.

H5I__mark_node() examines the supplied instance of H5I_id_info_t. If the force flag is set or if its ref count (info->count) is no greater than 1, it marks it for deletion (NOTE: it the app_ref flag is set, the ref count condition is changed to ref count - application ref count (info->app_count) <= 1).

If either of the above conditions are met, H5I__mark_node() discards the target of the void * that was provided on registration if a free function is provided for the type (The info->discard_cb is used for future objects if provided). The marked flag (info->marked) is then set to TRUE, and the id count (type_info→id_count) is decremented before H5I__mark_node() returns.

After the initial marking scan through the hash table associated with the target type, H5I_clear_type() sets the H5I_marking_g global to FALSE, and then uses HASH_ITER to setup a second scan, running the HASH_DELETE macro on every id whose marked flag was set in the prior scan. After removal from the hash table, each instance of H5I_id_info_t is freed via H5FL_FREE().

Assuming no errors have been detected, H5I_clear_type() then returns.

Multi-Thread safety concerns:

H5Iclear_type() has the multi-thread safety issues of accessing data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read/write access to the target instance of H5I_type_info_t, and
- Read/write/delete access to instance of H5I_id_info_t in the target index.
- Use of H5FL_FREE()

In addition, H5Iclear_type() displays a fundamental issue not seen so far in this pass through the public H5I API – specifically, H5I__mark_node() leaves index entries in a half deleted state pending their eventual full deletion in the second pass through the hash table. Absent changes in the algorithm, the only solution that comes to mind is to treat the entire H5Iclear_type() call as critical region.

From discussion with Dana, I gather that this mark and sweep approach was adopted to avoid issues with the regression test for H5Iitterate() (discussed below). Must investigate this to see if changes to H5I public API semantics would be required to avoid this issue.

```
/**
 * \ingroup H5IUD
 *
 * \brief Removes an identifier type and all identifiers within that type
 *
 * \param[in] type Identifier of identifier type which is to be destroyed
 *
 * \return \herr_t
 *
 * \details H5Idestroy_type deletes an entire identifier type \p type. All
 *          identifiers of this type are destroyed and no new identifiers of
 *          this type can be registered.
 *
 *          The type’s free function is called on all of the identifiers which
 *          are deleted by this function, freeing their memory. In addition,
 *          all memory used by this type’s hash table is freed.
 *
 *          Since the H5I_type_t values of destroyed identifier types are
 *          reused when new types are registered, it is a good idea to set the
 *          variable holding the value of the destroyed type to #H5I_UNINIT.
 *
 */
H5_DLL herr_t H5Idestroy_type(H5I_type_t type);

H5Idestroy_type()
 +-H5I__destroy_type()
    +-H5I_clear_type()
     | +-H5I__mark_node()
     |    +-H5I__mark_node()
     +-H5MM_xfree_const()
     |    +-H5MM_xfree()
     +-H5MM_xfree()
```

In a nutshell:

Discard the target index type, along with all IDs that may reside in the target index.

In more detail:

H5Idestroy_type() verifies that the target type is not an internal HDF5 library type, and then calls H5I__destroy_type(), returning whatever value that function returns.

H5I__destroy_type() validates the type id reading the H5I_next_type_g global in the process, and then gets a pointer to the target type from the indicated entry in the H5I_type_info_array_g global array of pointer to H5I_type_info_t. It verifies that this pointer is not NULL, and that type_info->init_count is positive.

If these tests pass, H5I__destroy_type() then calls H5I_clear_type() with the force parameter set to TRUE, and the app_ref parameter set to FALSE. Any error return from this call is ignored.

H5I_clear_type() is discussed in detail in H5Iclear_type() above. For purposes of this discussion, it should be sufficient to note that with the above parameters, it will discard all IDs of the target type.

On H5I_clear_type()'s return, H5I__destroy_type() tests to see if the H5I_CLASS_IS_APPLICATION flag is set in type_info->cls->flags. If so, it frees type_info→cls via a call to H5MM_xfree_const().

The hash table is then freed via the HASH_CLEAR() macro, followed by the *type_info itself (via H5MM_xfree()).

Finally, the target entry in the H5I_type_info_array_g global array is set to NULL just before the function returns.

Multi-Thread safety concerns:

Structurally, H5Idestroy_type() is very similar to H5Iclear_type(), and thus has all the multi-thread concerns surrounding that call with the addition of write access to the H5I_type_info_array_g global array.

```
/**
 * \ingroup H5IUD
 *
 * \brief Increments the reference count on an ID type
 *
 * \param[in] type The identifier of the type whose reference count is to be incremented
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Iinc_type_ref() increments the reference count on an ID type. The
 *          reference count is used by the library to indicate when an ID type
 *          can be destroyed.
 *
 *          The type parameter is the identifier for the ID type whose
 *          reference count is to be incremented. This identifier must have
 *          been created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Iinc_type_ref(H5I_type_t type);
```

```
H5Iinc_type_ref()
 +-H5I__inc_type_ref()
```

In a nutshell:

Increment the reference count on the indicated index.

In more detail:

After accessing the H5I_next_type_g global variable to validate the supplied type and verify that it is not a library type, H5Iinc_type_ref() calls H5I__inc_type_ref(), and returns whatever that function returns.

H5I__inc_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, increments type_info->init_count, and returns the new value of that field.

Multi-Thread safety concerns:

H5Iinc_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read/write access to the target instance of H5I_type_info_t, specifically the init_count field.

```
/**
 * \ingroup H5IUD
 *
 * \brief Decrements the reference count on an identifier type
 *
 * \param[in] type The identifier of the type whose reference count is to be decremented
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Idec_type_ref() decrements the reference count on an identifier
 *          type. The reference count is used by the library to indicate when
 *          an identifier type can be destroyed. If the reference count reaches
 *          zero, this function will destroy it.
 *
 *          The type parameter is the identifier for the identifier type whose
 *          reference count is to be decremented. This identifier must have
 *          been created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Idec_type_ref(H5I_type_t type);

H5Idec_type_ref()
 +-H5I_dec_type_ref()
      +-H5I_clear_type()
       | +-H5I__mark_node()
       |    +-H5I__mark_node()
       +-H5MM_xfree_const()
       |    +-H5MM_xfree()
       +-H5MM_xfree()
```

In a nutshell:

Decrement the reference count of the target type. If the index count drops to zero, discard all IDs in the target index, and then discard the index type as well.

In greater detail:

After verifying that the supplied type is not a library type, H5Idec_type_ref() calls H5I_dec_type_ref(), and returns whatever value that function returns.

Using the H5I_next_type_g global, H5I_dec_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, it tests to see if type_info→init_count is 1.

If it is not, it decrements type_info->init_count and returns that value.

If it is, it calls H5I__destroy_type() and returns zero.

H5I__destroy_type() is discussed in detail in H5Idestroy_type() above, and thus need not be discussed here.

Multi-Thread safety concerns:

As per H5Idestroy_type().

```
**
 * \ingroup H5IUD
 *
 * \brief Retrieves the reference count on an ID type
 *
 * \param[in] type The identifier of the type whose reference count is to be retrieved
 *
 * \return Returns the current reference count on success, negative on failure.
 *
 * \details H5Iget_type_ref() retrieves the reference count on an ID type. The
 *          reference count is used by the library to indicate when an ID type
 *          can be destroyed.
 *
 *          The type parameter is the identifier for the ID type whose
 *          reference count is to be retrieved. This identifier must have been
 *          created by a call to H5Iregister_type().
 *
 */
H5_DLL int H5Iget_type_ref(H5I_type_t type);
```

```
H5Iget_type_ref()
 +-H5I__get_type_ref()
```

In a nutshell:

Return the reference count on the target type.

In greater detail:

After accessing the H5I_next_type_g global variable to validate the supplied type and verify that it is not a library type, H5Iget_type_ref() calls H5I__get_type_ref(), and returns whatever value that function returns.

H5I__get_type_ref() does sanity checks on the supplied type. If they pass, it gets a pointer (type_info) to the target type from the indicated entry in the H5I_type_info_array_g global array. It verifies that this pointer is not NULL, and if so, it returns the current value of type_info->init_count.

Multi-Thread safety concerns:

H5Iget_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t, specifically the init_count field.

```
/**
 * \ingroup H5IUD
 *
 * \brief Finds the memory referred to by an ID within the given ID type such
 *        that some criterion is satisfied
 *
 * \param[in] type The identifier of the type to be searched
 * \param[in] func The function defining the search criteria
 * \param[in] key A key for the search function
 *
 * \return Returns a pointer to the object which satisfies the search function
 *         on success, NULL on failure.
 *
 * \details H5Isearch() searches through a given ID type to find an object that
 *          satisfies the criteria defined by \p func. If such an object is
 *          found, the pointer to the memory containing this object is
 *          returned. Otherwise, NULL is returned. To do this, \p func is
 *          called on every member of type \p type. The first member to satisfy
 *          \p func is returned.
 *
 *          The \p type parameter is the identifier for the ID type which is to
 *          be searched. This identifier must have been created by a call to
 *          H5Iregister_type().
 *
 *          The parameter \p func is a function pointer to a function which
 *          takes three parameters. The first parameter is a \c void* and will
 *          be a pointer to the object to be tested. This is the same object
 *          that was placed in storage using H5Iregister(). The second
 *          parameter is a hid_t and is the ID of the object to be tested. The
 *          last parameter is a \c void*. This is the \p key parameter and can
 *          be used however the user finds helpful, or it can be ignored if it
 *          is not needed. \p func returns 0 if the object it is testing does
 *          not pass its criteria. A non-zero value should be returned if the
 *          object does pass its criteria. H5I_search_func_t is defined in
 *          H5Ipublic.h and is shown below.
 *          \snippet this H5I_search_func_t_snip
 *          The \p key parameter will be passed to the search function as a
 *          parameter. It can be used to further define the search at run-time.
 *
 */
H5_DLL void *H5Isearch(H5I_type_t type, H5I_search_func_t func, void *key);


typedef struct {
    H5I_search_func_t app_cb;  /* Application's callback routine */
    void *            app_key; /* Application's "key" (user data) */
    void *            ret_obj; /* Object to return */
} H5I_search_ud_t;


/* User data for iterator callback for ID iteration */
typedef struct {
    H5I_search_func_t user_func;  /* 'User' function to invoke */
    void *            user_udata; /* User data to pass to 'user' function */
    hbool_t           app_ref;    /* Whether this is an appl. ref. call */
    H5I_type_t        obj_type;   /* Type of object we are iterating over */
} H5I_iterate_ud_t;


H5Isearch()
 +-H5I_iterate()
    +-H5I__iterate_cb()
       +-H5I__unwrap()
       |  +-H5VL_object_data()
       |  |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |  |     +- ??? -- must investigate
       |  +-H5T_get_actual_type()
       |     +-H5VL_object_data()
       |        +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |           +- ??? -- must investigate
       +-func() -- user function provided in call to H5Iserach()
```

In a nutshell:

Scan through the IDs of the target type, running the supplied search function on each. Return when the search function returns either success or error.

In greater detail:

On entry, H5Isearch() verifies that the supplied type is not a library type, and then initializes an instance of H5I_search_ud_t as follows:

```
udata.app_cb  = func;
udata.app_key = key;
udata.ret_obj = NULL;
```

It then calls H5I_iterate() with the supplied type, H5I__search_cb() as the func parameter, a pointer to the instance of H5I_search_ud_t as the udata parameter, and with the app_ref parameter set to TRUE. The return value of H5I_iterate() is ignored, and the function returns udata.ret_obj to the caller.

On entry, H5I_iterate() does some sanity checks, looks up the target type's instance of H5I_type_info_t in the global H5I_type_info_array_g, and stores that pointer in type_info.

If type_info is not NULL, type_info->init_count > 0, and type_info->id_count > 0, H5I_iterate() proceeds as follows:

First, it sets its user data in an instance H5I_iterate_ud_t and initializes it as follows:

```
iter_udata.user_func  = func;    // H5I__search_cb in this case
iter_udata.user_udata = udata;   // udata from H5Isearch()
iter_udata.app_ref    = app_ref; // TRUE in this case
iter_udata.obj_type   = type;    // the target index type
```

The the function uses the HASH_ITER macro to set up a for loop to iterate through all IDs in the target type. For each such ID that is not marked for deletion (i.e. the marked field in the associated instance of H5I_id_info_t is not set), H5I_iterate() calls H5I__iterate_cb() with the item parameter pointing to the instance of H5I_id_info_t associated with the current ID, NULL for the key parameter, and the udata parameter pointing to the instance of H5I_iterate_ud_t just initialized. If H5I_iterate() returns either H5_ITER_STOP or H5_ITER_ERROR, H5I_iterate() breaks out of the for loop and returns – flagging an error in the latter case.

H5I__iterate_cb() checks to see if the app_ref field of the user data provided by H5I_iterate() is TRUE, and if application reference count on the target instance of H5I_id_info_t (*_item) is positive.

If these tests pass, H5I__iterate_cb() calls H5I__unwrap() on the void pointer that was passed to H5Iregister(), and passes the result to the search function (instance of H5I_search_func_t) that was passed into H5Isearch(). If the search function returns a positive value, the return value of H5I__iterate_cb() is set to H5_ITER_STOP, if a negative value, H5_ITER_ERROR. Otherwise, H5I__iterate_cb() returns H5_ITER_CONT.

In the context of external API's, H5I__unwrap() is a no-op. It simply returns the void pointer that was passed to it in the object parameter.

However, for library IDs of H5I_FILE, H5I_GROUP, H5I_DATASET, or H5I_ATTR type, the void pointer is cast tor a pointer to H5VL_object_t, and passed to H5VL_object_data(). The return value of H5VL_object_data() is returned to the caller. Similarly, if the ID is of type H5I_DATATYPE, the void pointer is cast to a pointer to H5T_t and passed to H5VL_object_data() -- where
if may be passed to H5VL_object_data(). Again, the value from H5T_get_actual_type() is returned to the caller.

Tracing through H5VL_object_data() and its subsequent calls to see what is going on here is beyond the scope of the current investigation. However, this will have to be addressed when H5VL is investigated, if not before.

Multi-Thread safety concerns:

H5Isearch() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t.
- Read access to every instance of H5I_id_info_t In the target index

In addition, there are the unknown issues raised by H5I__unwrap() as discussed above.

Leaving aside the issues raised by H5I__unwrap(), H5I has little control over the behavior of the user provided search function.

Finally, there is the fact that H5Isearch() iterates through the IDs in the target index. While H5Isearch() doesn't have the obvious data consistency issues of H5Iclear() and H5Idestroy(), it will probably be convenient to apply the same solutions to it as well.

```
/**
 * \ingroup H5IUD
 *
 * \brief Calls a callback for each member of the identifier type specified
 *
 * \param[in] type The identifier type
 * \param[in] op The callback function
 * \param[in,out] op_data The data for the callback function
 *
 * \return The last value returned by \p op
 *
 * \details H5Iiterate() calls the callback function \p op for each member of
 *          the identifier type \p type. The callback function type for \p op,
 *          H5I_iterate_func_t, is defined in H5Ipublic.h as:
 *          \snippet this H5I_iterate_func_t_snip
 *          \p op takes as parameters the identifier and a pass through of
 *          \p op_data, and returns an herr_t.
 *
 *          A positive return from op will cause the iteration to stop and
 *          H5Iiterate() will return the value returned by \p op. A negative
 *          return from \p op will cause the iteration to stop and H5Iiterate()
 *          will return failure. A zero return from \p op will allow iteration
 *          to continue, as long as there are other identifiers remaining in
 *          type.
 *
 * \since 1.12.0
 *
 */
H5_DLL herr_t H5Iiterate(H5I_type_t type, H5I_iterate_func_t op, void *op_data);
```

```
typedef herr_t (*H5I_iterate_func_t)(hid_t id, void *udata);

typedef struct {
    H5I_iterate_func_t op;      /* Application's callback routine */
    void *             op_data; /* Application's user data */
} H5I_iterate_pub_ud_t;


H5Iiterate()
 +-H5I_iterate()
    +-H5I__iterate_cb()
       +-H5I__unwrap()
       |  +-H5VL_object_data()
       |  |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |  |     +- ??? -- must investigate
       |  +-H5T_get_actual_type()
       |     +-H5VL_object_data()
       |        +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |           +- ??? -- must investigate
       +-op() -- user function provided in call to H5Iiterate()
```

In a nutshell:

Scan through the IDs of the target type, running the supplied function on each. Return early if the supplied function returns either success or error.

In greater detail:

H5Iiterate() initializes an instance of H5I_iterate_pub_ud_t as follows:

```
     int_udata.op          = op;
     int_udata.op_data = op_data;
```

and then calls H5I_iterate() with the supplied type as the type parameter, H5I__iterate_pub_cb as the func parameter, the int_udata as the udata, and TRUE as the app_ref parameter. H5Iiterate() returns the value returned by H5I_iterate().

From this point, H5Iiterate is very similar to H5Isearch()

H5I_iterate() is the same as in H5Isearch() with the exception of the initialization of its instance of H5I_iterate_ud_t:

```
        iter_udata.user_func  = func;        // H5I__iterate_pub_cb
in this case
        iter_udata.user_udata = udata;   // udata from H5Iiterate()
-- this is
                                                              // the
delta
        iter_udata.app_ref    = app_ref;   // TRUE in this case
        iter_udata.obj_type   = type;        // the target index type
```

Similarly, H5I__iterate_cb() functions much as it does in H5Isearch(), the difference being in the user function called (H5I__iterate_pub_cb() vs H5I__search_cp()) and the user data.

H5I__iterate_pub_cb() is simpler than H5I__search_cb(). It just calls the function supplied to H5Isearch() with the current ID and udata supplied to H5Isearch() as parameters. It translates the return value to either H5_ITER_STOP, H5_ITER_ERROR, or H5I_ITER_CONT as appropriate, and returns.

Multi-Thread safety concerns:

Much the same as H5Isearch(), with additional concerns about the function supplied to H5Iiterate(). Since this function is only supplied with the ID of the index entry under examination, it will probably have to make a H5I call to obtain the associated data – complicating the multi-thread safety problem.

```
/**
 * \ingroup H5IUD
 *
 * \brief Returns the number of identifiers in a given identifier type
 *
 * \param[in] type The identifier type
 * \param[out] num_members Number of identifiers of the specified identifier type
 *
 * \return \herr_t
 *
 * \details H5Inmembers() returns the number of identifiers of the identifier
 *          type specified in \p type.
 *
 *          The number of identifiers is returned in \p num_members. If no
 *          identifiers of this type have been registered, the type does not
 *          exist, or it has been destroyed, \p num_members is returned with
 *          the value 0.
 *
 */
H5_DLL herr_t H5Inmembers(H5I_type_t type, hsize_t *num_members);

H5Inmembers()
 +-H5I_nmembers()
```

In a nutshell:

Return the number of IDs in the target type.

In greater detail:

After accessing the H5I_next_type_g and the H5I_type_info_array_g global variables to validate the supplied type and verify that it is not a library type, H5Inmembers() calls H5I_nmembers() to obtain the current number of entries in the target type, and return that value in Inum_members.

H5I_nmembers() validates the type again, and then obtains a pointer (type_info) to the target instance of H5I_type_info_t. If this pointer is NULL, or if type_info→init_count is non-positive, H5I_nmembers() returns zero. Otherwise, it returns type_info->id_count.

Multi-Thread safety concerns:

H5Iget_type_ref() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,
- Read access to the target instance of H5I_type_info_t, specifically the init_count and id_count fields.

```
/**
 * \ingroup H5IUD
 *
 * \brief Determines whether an identifier type is registered
 *
 * \param[in] type Identifier type
 *
 * \return \htri_t
 *
 * \details H5Itype_exists() determines whether the given identifier type,
 *          \p type, is registered with the library.
 *
 * \since 1.8.0
 *
 */
H5_DLL htri_t H5Itype_exists(H5I_type_t type);

H5Itype_exists()
```

In a nutshell:

Return TRUE if the specified type exists, and FALSE otherwise.

In greater detail:

Look up the entry in the global H5I_type_info_array_g array indicated by the supplied type. Return FALSE if this entry is NULL, and TRUE otherwise.

Multi-Thread safety concerns:

H5Itype_exists() accesses data structures that are visible to other threads – specifically:

- Read access to the H5I_type_info_array_t and H5I_next_type_g global variables,

```
/**
 * \ingroup H5I
 *
 * \brief Determines whether an identifier is valid
 *
 * \obj_id{id}
 *
 * \return \htri_t
 *
 * \details H5Iis_valid() determines whether the identifier \p id is valid.
 *
 * \details Valid identifiers are those that have been obtained by an
 *          application and can still be used to access the original target.
 *          Examples of invalid identifiers include:
 *          \li Out of range values: negative, for example
 *          \li Previously-valid identifiers that have been released:
 *              for example, a dataset identifier for which the dataset has
 *              been closed
 *
 *          H5Iis_valid() can be used with any type of identifier: object
 *          identifier, property list identifier, attribute identifier, error
 *          message identifier, etc. When necessary, a call to H5Iget_type()
 *          can determine the type of the object that \p id identifies.
 *
 * \since 1.8.3
 *
 */
H5_DLL htri_t H5Iis_valid(hid_t id);

H5Iis_valid()
 +-H5I__find_id()
    +-(id_info->realize_cb)()
    +-H5I__remove_common()
    +-(id_info->discard_db)()
```

In a nutshell:

Look up the supplied ID. If it doesn't exist, of if it has a zero application ref count (i.e. it is HDF5 library internal), return FALSE. Otherwise return TRUE.

In greater detail:

H5Iis_valid() calls H5I__find_id() to look up a pointer to the instance of H5I_id_info_t (info) associated with the supplied ID. See H5Iobject_verify() for a discussion of this call.

If info is NULL, or if info→app_count is zero, H5Iis_valid() returns FALSE. Otherwise, it returns TRUE.

Multi-Thread safety concerns:

See H5Iobject_verify().

```
/**
 * \ingroup H5I
 *
 * \brief Registers a "future" object under a type and returns an ID for it
 *
 * \param[in] type The identifier of the type of the new ID
 * \param[in] object Pointer to "future" object for which a new ID is created
 * \param[in] realize_cb Function pointer to realize a future object
 * \param[in] discard_cb Function pointer to destroy a future object
 *
 * \return \hid_t{object}
 *
 * \details H5Iregister_future() creates and returns a new ID for a "future" object.
 *          Future objects are a special kind of object and represent a
 *          placeholder for an object that has not yet been created or opened.
 *          The \p realize_cb will be invoked by the HDF5 library to 'realize'
 *          the future object as an actual object.  A call to H5Iobject_verify()
 *          will invoke the \p realize_cb callback and if it successfully
 *          returns, will return the actual object, not the future object.
 *
 * \details The \p type parameter is the identifier for the ID type to which
 *          this new future ID will belong. This identifier may have been created
 *          by a call to H5Iregister_type() or may be one of the HDF5 pre-defined
 *          ID classes (e.g. H5I_FILE, H5I_GROUP, H5I_DATASPACE, etc).
 *
 * \details The \p object parameter is a pointer to the memory which the new ID
 *          will be a reference to. This pointer will be stored by the library,
 *          but will not be returned to a call to H5Iobject_verify() until the
 *          \p realize_cb callback has returned the actual pointer for the object.
 *
 *          A  NULL value for \p object is allowed.
 *
* \details The \p realize_cb parameter is a function pointer that will be
 *          invoked by the HDF5 library to convert a future object into an
 *          actual object.   The \p realize_cb function may be invoked by
 *          H5Iobject_verify() to return the actual object for a user-defined
 *          ID class (i.e. an ID class registered with H5Iregister_type()) or
 *          internally by the HDF5 library in order to use or get information
 *          from an HDF5 pre-defined ID type.  For example, the \p realize_cb
 *          for a future dataspace object will be called during the process
 *          of returning information from H5Sget_simple_extent_dims().
 *
 *          Note that although the \p realize_cb routine returns
 *          an ID (as a parameter) for the actual object, the HDF5 library
 *          will swap the actual object in that ID for the future object in
 *          the future ID.  This ensures that the ID value for the object
 *          doesn't change for the user when the object is realized.
 *
 *          Note that the \p realize_cb callback could receive a NULL value
 *          for a future object pointer, if one was used when H5Iregister_future()
 *          was initially called.  This is permitted as a means of allowing
 *          the \p realize_cb to act as a generator of new objects, without
 *          requiring creation of unnecessary future objects.
 *
 *          It is an error to pass NULL for \p realize_cb.
 *
 * \details The \p discard_cb parameter is a function pointer that will be
 *          invoked by the HDF5 library to destroy a future object.  This
 *          callback will always be invoked for _every_ future object, whether
 *          the \p realize_cb is invoked on it or not.  It's possible that
 *          the \p discard_cb is invoked on a future object without the
 *          \p realize_cb being invoked, e.g. when a future ID is closed without
 *          requiring the future object to be realized into an actual one.
 *
 *          Note that the \p discard_cb callback could receive a NULL value
 *          for a future object pointer, if one was used when H5Iregister_future()
 *          was initially called.
 *
 *          It is an error to pass NULL for \p discard_cb.
 *
 * \note The H5Iregister_future() function is primarily targeted at VOL connector
 *          authors and is _not_ designed for general-purpose application use.
 *
 */
H5_DLL hid_t H5Iregister_future(H5I_type_t type, const void *object,
                               H5I_future_realize_func_t realize_cb,
                               H5I_future_discard_func_t discard_cb);

H5Iregister_future()
 +-H5I__register()
```

In a nutshell:

H5Iregister_future() inserts the supplied void pointer in the index of the indicated type, marks the entry as a future object, and decorates it with the supplied realize and discard callbacks. The function returns an ID that can be used to access this void pointer (or its realized version) at a later date.

In more detail:

Test to see if the supplied type is a library type (i.e. one used internally. Fail if it is. Similarly, verify that the realize and discard callbacks are defined. If all tests pass, call H5I__register() with the app_ref. realize_cb, and discard_cb parameters set to TRUE, and the supplied values respectively.

Further processing is as per H5Iregister(), with the exception that the is_future flag is set to TRUE, not FALSE.

Multi-Thread safety concerns:

As per H5Iregister().

8 Appendix 2 – H5I private API calls

In addition to its public and developer APIs, H5I also has a private API providing indexing services to the HDF5 library. For the most part, these calls are similar to their cognates in the public API – but there are some differences, and also some calls which offer additional capabilities.

Since the objective of this exercise is make H5I multi-thread safe so it can be safely called by multiple threads in multi-thread safe VOL connectors, at first glance, the internal H5I API is not relevant to this effort. However, the internal H5I API is used by other packages – including those necessary to support multi-thread VOL connectors.

The list of internal H5I API calls below is taken from H5Iprivate.h. Most entries are annotated with a reference to the relevant public API call. Those with no public API cognate have more extensive annotations.

```
H5_DLL herr_t     H5I_register_type(const H5I_class_t *cls);
```

See H5Iregister_type()

```
H5_DLL int64_t    H5I_nmembers(H5I_type_t type);
```

See H5Inmembers()

```
H5_DLL herr_t     H5I_clear_type(H5I_type_t type, hbool_t force, hbool_t app_ref);
```

See H5Iclear_type()

```
H5_DLL H5I_type_t H5I_get_type(hid_t id);
```

See H5Iget_type()

```
H5_DLL herr_t     H5I_iterate(H5I_type_t type, H5I_search_func_t func, void *udata, hbool_t
app_ref);
```

See H5Isearch() and H5Iiterate()

```
H5_DLL int        H5I_get_ref(hid_t id, hbool_t app_ref);
```

See H5Iget_ref()

```
H5_DLL int        H5I_inc_ref(hid_t id, hbool_t app_ref);
```

See H5Iinc_ref()

```
H5_DLL int        H5I_dec_ref(hid_t id);
```

```
H5I_dec_ref()
 +-H5I__dec_ref()
    +-H5I__find_id()
    |  +-(id_info->realize_cb)()
    |  +-H5I__remove_common()
    |  +-(id_info->discard_db)()
    +-(type_info->cls->free_func)((void *)info->object, request)
    +-H5I__remove_common()
```

H5I_dec_ref() verifies that the ID is non-negative, and then calls H5I__dec_ref() with H5_REQUEST_NULL as the request parameter.

See H5Idec_ref() for further details.

```
H5_DLL int        H5I_dec_app_ref(hid_t id);
```

See H5Idec_ref()

```
H5_DLL int        H5I_dec_app_ref_async(hid_t id, void **token);

H5I_dec_app_ref_async()
 +-H5I__dec_app_ref()
    +-H5I__dec_ref()
    |  +-H5I__find_id()
    |  |  +-(id_info->realize_cb)()
    |  |  +-H5I__remove_common()
    |  |  +-(id_info->discard_db)()
    |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  +-H5I__remove_common()
    +-H5I__find_id()
       +-(id_info->realize_cb)()
       +-H5I__remove_common()
       +-(id_info->discard_db)()
```

H5I_dec_app_ref_async() verifies that the ID is non-negative, and then calls H5I__dec_app_ref(). This differs from calls to H5I__dec_app_ref() elsewhere in that the request parameter passed into H5I__dec_app_ref() is user supplied, and not hard coded to H5_REQUEST_NULL as in H5Idec_ref() above. This request appears to be passed into the free function from the class when it is called on the void pointer that was passed in on ID registration. It does not appear to be used elsewhere.

Otherwise, the call to H5I__dec_app_ref() seems to be as described in H5Idec_ref() above.

```
H5_DLL int        H5I_dec_app_ref_always_close(hid_t id);

H5I_dec_app_ref_always_close()
 +-H5I__dec_app_ref_always_close()
    +-H5I__dec_app_ref()
    |  +-H5I__dec_ref()
    |  |  +-H5I__find_id()
    |  |  |  +-(id_info->realize_cb)()
    |  |  |  +-H5I__remove_common()
    |  |  |  +-(id_info->discard_db)()
    |  |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  |  +-H5I__remove_common()
    |  +-H5I__find_id()
    |     +-(id_info->realize_cb)()
    |     +-H5I__remove_common()
    |     +-(id_info->discard_db)()
    +-H5I_remove()
       +-H5I__remove_common()
```

H5I_dec_app_ref_always_close() verifies that the supplied ID is non-negative, and then calls H5I__dec_app_ref_always_close() with the supplied ID and the request parameter set to H5_REQUEST_NULL. It returns whatever value H5I__dec_app_ref_always_close() returns.

After initial sanity checks, H5I__dec_app_ref_always_close() calls H5I__dec_app_ref() with the supplied id and request (H5_REQUEST_NULL in this case). See H5Idec_ref() for a discussion of H5I__dec_app_ref() under these circumstances.

When H5I__dec_app_ref(), H5I__dec_app_ref_always_close() checks for failure, and calls H5I_remove() if a failure is detected. This appears to be an attempt to force removal of the ID even if the free call fails -- see the following comment:

```
    /*
     * If an object is closing, we can remove the ID even though the free
     * method might fail.  This can happen when a mandatory filter fails to
     * write when a dataset is closed and the chunk cache is flushed to the
     * file.  We have to close the dataset anyway. (SLU - 2010/9/7)
     */
```

H5I_remove() is discussed in H5Iremove_verify() above.

```
H5_DLL int        H5I_dec_app_ref_always_close_async(hid_t id, void **token);

H5I_dec_app_ref_always_close_async()
 +-H5I__dec_app_ref_always_close()
    +-H5I__dec_app_ref()
    |  +-H5I__dec_ref()
    |  |  +-H5I__find_id()
    |  |  |  +-(id_info->realize_cb)()
    |  |  |  +-H5I__remove_common()
    |  |  |  +-(id_info->discard_db)()
    |  |  +-(type_info->cls->free_func)((void *)info->object, request)
    |  |  +-H5I__remove_common()
    |  +-H5I__find_id()
    |     +-(id_info->realize_cb)()
    |     +-H5I__remove_common()
    |     +-(id_info->discard_db)()
    +-H5I_remove()
       +-H5I__remove_common()
```

As per H5I_dec_app_ref_always_close() above, save that H5I_dec_app_ref_always_close_async() takes a token parameter (void **), that is passed down to +-H5I__dec_app_ref_always_close() as its request parameter. This parameter is eventually passed to the free function for the type if the ref count drops to zero.

```
H5_DLL int        H5I_dec_type_ref(H5I_type_t type);
```

See H5Idec_type_ref()

```
H5_DLL herr_t     H5I_find_id(const void *object, H5I_type_t type, hid_t *id /*out*/);


/* User data for iterator callback for retrieving an ID corresponding to an object pointer
*/
typedef struct {
    const void *object;   /* object pointer to search for */
    H5I_type_t  obj_type; /* type of object we are searching for */
    hid_t       ret_id;   /* ID returned */
} H5I_get_id_ud_t;


H5I_find_id()
 +-H5I__find_id_cb()
    +-H5I__unwrap()
       +-H5VL_object_data()
       |  +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
       |     +- ??? -- must investigate
       +-H5T_get_actual_type()
          +-H5VL_object_data()
             +-(vol_obj->connector->cls->wrap_cls.get_object)(vol_obj->data)()
                +- ??? -- must investigate
```

In a nutshell:

Scan all IDs in the target type. If an ID has a void pointer associated with it that matches the supplied void *, return this ID in *id.

In greater detail:

After initial sanity checks, and setting *id = H5I_INVALID_HID, H5I_find_id() looks up the instance of H5I_type_info_t associated with the supplied type in the H5I_type_info_array_g global array. If the target type exists and has been initialized, and has at least one entry. the function initializes an instance of H5I_get_id_ud_t as follows:

```
    /* Set up iterator user data */
    udata.object   = object;
    udata.obj_type = type;
    udata.ret_id   = H5I_INVALID_HID;
```

and then uses the HASH_ITER uthash macro to set up a for loop that visits each ID in the target type.

It calls H5I__find_id_cb() on each such entry, returning an error if H5I__find_id_cb() returns and error, and breaking out of the for loop if H5I__find_id_cb() returns H5_ITER_STOP. In either case, H5I_find_id() sets *id = udata.ret_id after it exits the for loop.

H5I__find_id_cb() calls H5I__unwrap() on the void pointer associated with the target id. It tests to see if the return value of H5I__unwrap() equals udata->object. If it does, it sets udata->ret_id = info->id, and returns H5_ITER_STOP.

See H5Isearch() above for a discussion of H5I__unwrap(). The bottom line is that it makes calls into H5VL, and then into a VOL connector callback -- with the resulting potential multithread safety issues. I am putting this issue to one side pending review of H5VL.

```
/* NOTE:    The object and ID functions below deal in non-VOL objects (i.e.;
 *          H5S_t, etc.). Similar VOL calls exist in H5VLprivate.h. Use
 *          the H5VL calls with objects that go through the VOL, such as
 *          datasets and groups, and the H5I calls with objects
 *          that do not, such as property lists and dataspaces. Datatypes
 *          are can be either named, where they will use the VOL, or not,
 *          and thus require special treatment. See the datatype docs for
 *          how to handle this.
 */

/* Functions that manipulate objects */
H5_DLL void * H5I_object(hid_t id);
```

See H5Iget_type()

```
H5_DLL void * H5I_object_verify(hid_t id, H5I_type_t type);
```

See H5Iobject_verify()

```
H5_DLL void * H5I_remove(hid_t id);
```

See H5Iremove_verify()

```
H5_DLL void * H5I_subst(hid_t id, const void *new_object);

H5I_subst()
  +-H5I__find_id()
     +-(id_info->realize_cb)()
     +-H5I__remove_common()
     +-(id_info->discard_db)()
```

In a nut shell:

Replace the void * associated with the ID with the supplied void *, returning the old void *.

In greater detail:

H5I_subst() calls H5I__find_id() to obtain the instance of H5I_id_info_t associated with the target ID. See H5Iobject_verify() above for a discussion of H5I__find_id().

Assuming that H5I__find_id() is successful, H5I_subst() sets info->object = new_object, and returns the original value of info->object.

```
H5_DLL htri_t H5I_is_file_object(hid_t id);

H5I_is_file_object()
```

```
 +-H5I_get_type()
 +-H5I_object()
 |  +-H5I__find_id()
 |     +-(id_info->realize_cb)()
 |     +-H5I__remove_common()
 |     +-(id_info->discard_db)()
 +-H5T_is_named()
```

H5I_is_file_object() calls H5I_get_type() to obtain the type of the supplied ID. If the type is either H5I_DATASET, H5I_GROUP, or H5I_MAP, the function return TRUE.

If the type is H5I_DATATYPE, it calls H5I_object() to obtain the instance of H5T_t associated with the id, calls H5T_is_named() on this instance, and returns whatever H5T_is_named() returns.

Otherwise, the function returns FALSE.

H5I_get_type() invokes the H5I_TYPE() macro to extract the type, and returns this value.

See H5Iobject_verify() for a discussion of H5I_object().

H5T_is_named() returns TRUE iff the datatype is named/committed. This is determined by examining fields of the supplied instance of H5T_t, so no special multi-thread issues beyond the usual race conditions.

```
/* ID registration functions */
H5_DLL hid_t  H5I_register(H5I_type_t type, const void *object, hbool_t app_ref);

H5I_register()
 +-H5I__register()
```

After some sanity checks H5I_register() calls H5I__register() with its parameters, and NULL for the realize_cb and discard_cp parameters.

See H5Iregister() above for a discussion of H5I__register()

```
H5_DLL herr_t H5I_register_using_existing_id(H5I_type_t type, void *object, hbool_t
app_ref,
                                             hid_t existing_id);

H5I_register_using_existing_id()
 +-H5I__find_id()
    +-(id_info->realize_cb)()
    +-H5I__remove_common()
    +-(id_info->discard_db)()
```

In a nutshell:

Register the supplied void * in the specified index with the specified ID. The function will fail if the ID is already in use, or if it doesn't belong to the specified index.

In greater detail:

H5I_register_using_existing_id() first verifies that the supplied id is not in use via a call to H5I__find_id(). (see H5Iobject_verify() for a discussion of H5I__find_id()). It then verifies that the supplied type is valid, and that the supplied id belongs to the supplied type. As part of these sanity checks, it looks up the instance of H5I_type_info_t associated with the supplied type and stores its address in type_info. In so doing, it reads the global H5I_next_type_g and the global H5I_type_info_array_g array.

If all these sanity checks pass, the function allocates an instance of H5I_id_info_t via H5FL_CALLOC() storing its address in info. (note the thread safety issue), initializes it, and then uses the uthash HASH_ADD() to insert it into the hash table of the specified index with the specified id. Before exiting, it sets type_info->last_id_info = info.

From a multi-thread safety perspective, H5I_register_using_existing_id() seems to have the same issues as H5I__register().

```
/* Debugging functions */
H5_DLL herr_t H5I_dump_ids_for_type(H5I_type_t type);
```

This is a debugging function, and thus can be skipped for now.

10 Appendix 3 – Fields modified by API Calls

The following table lists the fields in the relevant instances of H5I_type_info_t and H5I_id_info_t that are modified by the listed API calls. Note that this data is derived from inspection of the code, and thus some errors should be expected.

Note also that H5Iget_file_id() and H5Iget_name() have been omitted from this table.

Operation(s)	Field modified in H5I_type_info_t	Fields modified in H5I_id_info_t

H5Iregister()
H5Iregister_future()
H5I_register()
	nextid,
	id_count,
	last_id_info
	hash_table
		all – allocate instance and initialize

H5Iobject_verify()
H5I_object_verify()
H5I_object()
(no future objects)
	last_id_info	none

H5Iobject_verify()
H5I_object_verify()
H5I_object()
(future objects possible)
	last_id_info,
	hash_table,
	id_count
		all
		(delete one index entry, modify another)

H5Iremove_verify()
H5I_remove()
	hash_table,
	last_id_info,
	id_count
		all (free instance) or marked (if H5I_marking_g is TRUE)

H5Iget_type()
H5I_get_type()
(no future objects)
	none	none

H5Iget_type()
H5I_get_type()
(future objects possible)
	hash_table,
	id_count
		all
		(delete one index entry, modify another)

H5Iinc_ref()
H5I_inc_ref()
(no future objects)
	none	count,
		app_count

H5Iinc_ref()
H5I_inc_ref()
(future objects possible)
	hash_table,
	id_count
		all
		(delete one index entry, modify another)

H5Idec_ref()
H5I_dec_ref()
H5I_dec_app_ref_async()
H5I_dec_app_ref_always_close()
H5I_dec_app_ref_always_close_async()
	hash_table,
	last_id_info,
	id_count
		all

H5Iget_ref()
H5I_get_ref()
(no future objects)
	none	none

H5Iget_ref()
H5I_get_ref()
(future objects possible)
	hash_table,
	id_count
		all
		(delete and free one index entry, modify another)

H5Iregister_type	all – allocate and initialize	none

H5Iclear_type()
H5I_clear_type()
	hash_table,
	id_count
		all – delete and free most entries

H5Idestroy_type()	all – de-allocate when done	All – delete and free all entries
H5Iinc_type_ref()	init_count	none

H5Idec_type_ref()
H5I_dec_type_ref()
	just init_count if it remains positive, or all with de-allocation when done if init_count drops to zero	none, if init_count remains positive. otherwise all – delete and free all entries

H5Iget_type_ref()	none	none

H5Isearch()
H5Iiterate()
H5I_iterate()
H5I_find_id()
	potentially hash_table,
	id_count,
	last_id_info
		potentially all

H5Inmembers()
H5I_nmembers()
	id_count	none

H5Itype_exists()	none	none
H5Iis_valid() (no future objects)	none	none
H5Iis_valid() (future objects possible)	hash_table, id_count	all (delete one index entry, modify another)
H5I_subst() (no future objects)	last_id_info	object
H5I_subst() (future objects possible)	last_id_info, hash_table, id_count	all (delete one index entry, modify another)
H5I_is_file_object() (no future objects)	none	none
H5I_is_file_object() (future objects possible)	hash_table, id_count	all

		(delete one index entry, modify another)
H5I_register_using_existing_id() (no future objects)	id_count, last_id_info hash_table	all – allocate instance and initialize
H5I_register_using_existing_id() (future objects possible)	id_count, last_id_info hash_table	all – either allocate an instance and initialize, or delete one index entry, modify another