Making H5I Multi-Thread Safe:

A Sketch Design

John Mainzer

Lifeboat, LLC

8/15/22

# Introduction

Unlike H5E, it was obvious from the start that H5I would have at least one dependency – specifically H5E as H5I can flag errors. In principle, there seems no reason why H5I should have any further dependencies on any other packages in the HDF5 library. As shall be seen, principle and actuality can be two very different things.

As this is a very preliminary version of the sketch design for modifying H5I to be multi-thread safe, the initial objective is to outline the issues to be addressed, and to suggest possible directions for solutions. As shall be seen, it will be necessary to study additional packages before this sketch design can be brought into anything approaching a final form.

# A Quick Overview of H5I

H5I exists to provide indexing services 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, data sets, 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 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.
* Iteration and searching through indexes. Here the user provides a function and an index type. The function is executed on every entry in the target index (in the case of iteration) or until it reports success (in the case of searches).
* Miscellaneous services including incrementing and decrementing reference counts on entries and types of indexes, tests for validity, fetching the number of index entries of a give 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 H5I. The hope is that these dependencies can be resolved through re-architecting – but we will not know until the target packages have been examined in greater detail.

# Multi-Thread Issues in H5I

As with H5E, leaving aside the issue of the unexpected dependencies, there appear to be no fundamental reason 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.

## Use of other HDF5 packages in H5E

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 principle, this can be solved by either avoiding locking in H5I, or by dropping all locks before calling H5E.

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. 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. Initial scans suggest that there are no major multi-thread issues here. Further, from first principles, it should be possible to re-architect H5I so as to store this data with the relevant IDs and recover it within H5I if necessary. That said, the current focus is on H5I proper – thus this issue must be considered as a known unknown for now.

## Multi-thread thread issues in H5I proper

A review of the H5I public APIs reveals the following multi-thread issues in the current H5I implementation[[1]](#footnote-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. However, if HDF5 is going to stay with uthash for any length of time, this oversight should be corrected.

### Use of global tables and 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.

Locks around the appropriate critical regions are an obvious solution, but they present lock ordering issues in the event of failure. Again, locks can be dropped prior to error calls, but a solution based on atomic operations would be preferable.

### Potential race conditions in some structures

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 type of indexes) counts of existing IDs.

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

### Mark and Sweep Operations

H5I allows scans and searches of types of indexes. Our 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 leave IDs 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 unsatisfactory in a multi-thread implementation. Repairing this may require subtle modifications to the H5I API.

### 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 a real 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. Alternatively, once there is multi-thread support, this addition to the API may no longer be required.

### 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 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.

# Solutions

<in progress>

# Appendix 1 – H5I public API calls

<in progress>

# Appendix 2 – H5I internal API calls

<in progress>

1. A review of the internal H5I API is in progress. It may reveal further issues. [↑](#footnote-ref-1)