Unsafe Multithreaded H5I Iteration

Matt Larson

October 2024

1 Introduction

Testing of the multi-threaded H5VL module, while using H5F routines that delay global mutex acquisition until the native VOL layer, has revealed crashes that appear to be due to multi-threaded ID iteration not dealing with concurrent ID release safely.

2 Problem Overview

At the H5I level, multi-threaded ID iteration is performed within the following routines: H5I_iterate_internal(), H5I_get_first(), H5I_get_next(), and H5I_find_id(). All of these routines except H5I_iterate_internal() fundamentally work the same way and exhibit the same problem, but this discussion focuses on H5I_get_first() for clarity.

ID iteration works as follows:

1. If the type is non-empty, retrieve its first entry from the lock free hash table with `lfht_get_first()`.
2. Load the ID kernel.
3. If the kernel is marked for deletion, move on to the next ID via lfht_get_next()
4. If the kernel is not marked for deletion, unwrap its object value, then return that buffer and the ID itself to the caller.

IDs that are marked for deletion are skipped, in order to prevent an ID with a reference count of one from being released out from under this operation by a concurrent call to H5I_dec_ref().

However, this check doesn't seem to be sufficient. The other thread that owns the sole reference to the ID can interrupt the iteration, so long as it frees the ID and sets the marked flag *after* that check, and *before* H5I__unwrap(). (See appendix A for a summary of how ID release works.) This results in a memory error when the object buffer, released by the ID type's free function, is introspected by H5I__unwrap(). Some kind of change is necessary to make ID iteration multi-thread safe.

3 Potential Solutions

3.1 Increment the reference count of the ID retrieved during iteration

The root issue seems to be that the iteration routines, however they are used, result in a dangerous region where two threads hold a reference to an ID with a reference count of one.

Incrementing the reference count of the ID as soon as it is acquired from the lock free hash table, and decrementing it before advancing to the next ID, would eliminate the possibility for a concurrent free entirely. (H5I_inc_ref() safely handles the ID being freed out from under it due to use of the do not disturb lock and the ID free list.)

The primary downside of this approach is that the need to increment and decrement the ref count of each ID traversed, with each operation potentially taking multiple tries, might slow down ID iteration substantially.

This approach would also require clients to ID iteration to change how they use the routines, in most cases removing a call to H5I_inc_ref() right after the iteration.

H5I_get_first(), H5I_get_next(), and H5I_find_id are only used in a few places at present, and reworking their usage like this should be low effort. H5I_iterate() does not return an ID to the client, so modifying its reference counting behavior would have no effect on its clients.

3.2 Set 'do not disturb' flag during iteration

Setting the do not disturb flag during iteration would prevent concurrent interruption by H5I__dec_ref().

This would change the semantic meaning of the do not disturb flag, since it is intended to indicate concurrent modification, but it would prevent the invalid memory access.

This is the method used by H5I__iterate_internal() to handle executing potentially non-threadsafe user callbacks on IDs during iteration. While it suffices to prevent invalid memory accesses, this part of the multi-threaded H5I design is temporary and planned to be replaced, so bringing this behavior into other iteration methods may not be a satisfactory solution.

A ID Release Overview

H5I__dec_ref() works as follows:

1. Retrieve the target ID's value from the type via `H5I__find_id()`.
2. If it is already marked for deletion, fail this operation.
3. If the do not disturb flag is set, wait until it is unset
4. If the reference count of the id > 1 or there is no custom free callback, this operation can be rolled back. As such:
   (a) If the reference count is greater than one, simply set up the modified kernel to decrement the reference count.
   (b) Otherwise, if this ID will be freed and the free function is undefined, set up the modified kernel as marked for deletion.
   (c) Write the modified kernel in a single-shot compare-and-swap.
5. If the reference count $= 1$ **and** there is a custom free callback, this cannot be rolled back. As such:
   (a) Set the do not disturb flag on the kernel with a strong compare-and-swap.
   (b) Reload the kernel.
   (c) Potentially under the global mutex, execute the free function for the ID class. This may or may not release the object buffer under the ID, or any resources held under that object.
   (d) With a single-shot strong compare-and-swap, write that the ID is marked for deletion, and unset the do not disturb lock.
6. If the ID was marked for deletion as the result of previous steps, delete it from the lock free hash table and discard its ID info.

We can divide ID reference decrements into 3 cases:

1. The reference count > 1, so the ID is not freed. We can ignore this case.
2. The reference count $= 1$ and the ID type has no free function defined. In this case, there is a region where the ID in the type is marked for deletion, with the do not disturb flag unset. Due to no free function being defined, the object buffer in the kernel should remain valid memory, and a concurrent iteration should not be able to access any invalid memory. (The ID itself containing invalid memory should be prevented by the free list.)
3. The reference count $= 1$ and the ID type has a free function defined. Again, there is a region where the ID in the type is marked for deletion, with the do not disturb flag unset. `free_func()` has potentially freed the object buffer or a resource it references, and if this decrement interrupted a concurrent iteration, the iteration routine may attempt to access that invalid memory.