Unsafe Multithreaded H5I Iteration

Matt Larson

January 10, 2025

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, should 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.)

3.1.1 Problems with re-entrance and the do not disturb lock

If the increment/decrement reference count calls are directly added to iteration routines with no other changes, then a single thread will be able to deadlock itself waiting for the do not disturb flag to be released during recursive calls to increment/decrement reference count. This is because recursive user-defined callbacks internally result in increment/decrement reference count calls while the do not disturb lock is set on a given ID.

Other places in the current MT implementation of H5I assume that if the global mutex is held by this thread, and the target ID kernel's "have global mutex" flag is set, then the current thread is the one that set the do not disturb flag, and the do not disturb flag can be ignored since this must be a recursive case.

If another modification to H5I is made, allowing increment/decrement reference count calls to bypass the do not disturb lock using this logic, then flaws in the recursive check are revealed. Since the decrement reference count routine does not acquire the global mutex, it can set the do not disturb flag while another thread holds the global mutex. That other thread can then find that the global mutex is held, and that the target ID's "have global mutex" flag is set, and will consequently assume that it can ignore the do not disturb flag, even though this is not a recursive case.

The following is a specific example of this problem.

- Thread 1 begins H5Fclose on a file ID with only one reference.
- Threads 2 begins H5Dclose on a dataset ID with only one reference. It does not matter whether the dataset is in the file being closed by the other thread.
- Thread 2 sets the do not disturb lock on the dataset ID, and sets the do not disturb flag since the dataset ID call is not considered MT-safe, but does not yet acquire the global mutex.
- Thread 1 acquires the global mutex in order to invoke the file ID free callback. The file ID free callback eventually invokes H5D_flush_all(), which iterates over dataset IDs
- Thread 1 iterates over the dataset ID being operated on by Thread 2, bypasses the do-not-disturb flag since Thread 1 holds the global mutex and the dataset ID's "have global mutex" flag is set, and increments its reference count
- Thread 2 throws an assertion failure due to having a compare-and-swap fail during a region that should be protected by the do not disturb flag.

The most straightforward solution seems to be to replace or update the kernel's have global mutex flag with a thread ID that can be used for re-entrance checks instead, as described in the documentation for the ID kernel.

Additionally, multiple routines should preserve the value of the do not disturb flag and (if it remains after the redesign) the have global mutex flag in the event that the ID already has these fields set on entry. Otherwise, the fields will be unexpectedly unset during recursion. This applies to `H5I__dec_ref()` and `H5I__find_id()`, and may apply to `H5I__remove_common()` and `H5I__mark_node()` if it is considered valid for those routines to be used on recursive access to a single ID within a thread.

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.