Introduction:

This document justifies introduction of a new callback function to verify if attribute ID can be deleted. We are also considering changing parameters list to the existing callback instead of adding new function.

The document will be added to the multi-threaded H5A design document when new functionality is finalized.

```
/**
 * \ingroup H5A
 *
 * \brief Frees an attribute object when its ID reference count reaches zero.
 *
 * \param[in] attr_vol_obj Pointer to the VOL object wrapping the attribute
 * \param[in,out] request   Optional pointer to an async request (unused in sync case)
 *
 * \return \herr_t SUCCEED/FAIL
 *
 * \details H5A__close_cb() is the registered free_func for the H5I_ATTR ID type.
 *          It is automatically invoked by the H5I subsystem when an attribute ID's
 *          reference count drops to zero. This callback performs two operations:
 *          1. Calls H5VL_attr_close() to close the attribute through the VOL connector.
 *          2. Calls H5VL_free_object() to release the underlying VOL object structure.
 *
 *          These operations depend on the VOL connector being in a consistent state and
 *          properly configured. They may fail, for example, if the connector is in the process
 *          of finalizing, if required internal callbacks (such as attr_cls.close) are not defined,
 *          or if the VOL object structure is invalid (e.g., already on the free list).
 *          In a lock-free multi-threaded design, once the free function is called, it must succeed
 *          unconditionally – there is no rollback or retry at that level, so failure is not acceptable.
 *          To guarantee safety, this function is only invoked after a successful pre-check
(H5A__ff_precheck)
 *          verifies that deletion can proceed. If the pre-check fails, the actual deletion is skipped
 *          and deferred (the object remains on a free list to be cleaned up later when safe).
 *
 *          This function should not be called directly by user code; it is an internal callback.
 */
static herr_t H5A__close_cb(H5VL_object_t *attr_vol_obj, void **request);
```

```
static herr_t H5A__close_cb(H5VL_object_t *attr_vol_obj, void **request);
```

```
  H5A__close_cb()
  +- H5VL_attr_close()
  |    +- H5VL__attr_close()
  |         +- (cls->attr_cls.close)()
```

```
|            +- H5A__close()
|                +- H5O_close()
|                |  +- H5F_try_close()
|                |  +- H5O_loc_free()
|                +- H5A__shared_free()
|                +- H5G_name_free()
+- H5VL_free_object()
    +- H5VL_conn_dec_rc()
       +- H5I_dec_ref()
          +- H5I_enter()
          +- H5I_dec_ref_internal()
          |  +- H5I__dec_ref()
          |     +- H5TS_have_mutex()
          |     +- H5I__find_id()
          |     |  +- H5TS_have_mutex()
          |     |  +- lfht_find()
          |     |  +- (id_info_ptr->realize_cb)()
          |     |  +- H5I__remove_common()
          |     |  +- (id_info_ptr->discard_cb)()
          |     +- type_info_ptr->cls->free_func()
          |     +- lfht_delete()
          |     +- H5I__discard_mt_id_info()
          +- H5I__exit()
```

In a Nutshell:

H5A__close_cb closes and frees an attribute's VOL object when its ID reference count drops to zero. It finalizes the attribute through the VOL connector, then releases the H5VL_object_t wrapper that was holding the attribute. In a lock-free environment, this callback is only executed when it is guaranteed to succeed — a pre-check function ensures that the VOL object and connector are in a valid state beforehand. If the pre-check indicates the object cannot be safely closed (for instance, the VOL connector is shutting down or the object is already on a pending free list), H5A__close_cb is not called at that time. Instead, the deletion is deferred until it can be done safely. This design ensures that once H5A__close_cb runs, it will complete successfully, preserving the requirements of lock-free memory management.

In More Detail:

H5A__close_cb() is the free_func registered for attribute IDs. Whenever an attribute ID's reference count drops to zero (for example, after the user calls H5Aclose or an attribute ID is otherwise released), the H5I subsystem invokes this callback to dispose of the attribute. The function receives a pointer to an H5VL_object_t (the VOL object wrapper) that was associated with the attribute ID. The H5VL_object_t contains important context: a pointer to the VOL

connector used for this object (vol_obj->connector), a connector-specific object pointer (vol_obj->data internal to the connector), and an atomic reference counter (vol_obj->rc) for the object. The role of H5A__close_cb is to finalized the attribute and free the VOL object wrapper. This happens in two primary steps:

The function starts by closing the attribute via VOL by calling H5VL_attr_close() which forwards the close operation into the VOL layer. That call leads to H5VL__attr_close() which validates the object and then invokes the VOL connector's attribute-close callback, and in the case of the native HDF5 VOL connector, leads to a call to the internal H5A__close() routine, which frees the attribute's resources: it writes any cached metadata (if needed) to the file, closes the attribute's object header (which may in turn decrement file reference and attempt file closure if it was the last open object in the file), and frees memory associated with the attribute such as its shared components and group path name info. If the VOL connector is in a normal, consistent state, H5VL_attr_close() should succeed. However, if the connector is not properly configured or is in the process of shutting down, this step could fail. For example, if the VOL connector's attr_cls.close function pointer is NULL (meaning the connector does not support attribute closing), or if the connector has already been terminated, H5VL_attr_close() would return an error. Under typical operation, connectors are required to provide this callback, so a missing attr_close is generally treated as an internal error rather than a runtime issue.

If the attribute close succeeds, the next step is to free the VOL object wrapper via H5VL_free_object(). This routine will decrement the reference count on the H5VL_object_t and free its memory if that count drops to zero. In a single-threaded scenario, the reference count (rc) for the VOL object is usually one when an object is no longer in use (since the ID that referenced it is about to be closed). Therefore, H5VL_free_object() will typically free the wrapper and then call H5VL_conn_dec_rc(), which decrements the usage count of the VOL connector itself. Ultimately, this may trigger an H5I_dec_ref() on the connector's ID which unregisters the connector if no more objects are using it). For instance, if this attribute was the last object using a given VOL plugin, the plugin's reference count would drop to zero, causing the connector to close and unload. During these cleanup steps, failures could in theory occur too - e.g., if the VOL connector ID was somehow invalid or if the connector's shutdown routines encounter an error. In normal operation, these are rare; they would indicate a serious issue such as a connector that was already unregistered or a bug in the ref counting logic.

This tight dependency on a functioning VOL connector means H5A__close_cb() cannot be called blindly in a lock-free environment. Once the H5I subsystem decides to delete an ID, it must be certain that the deletion will succeed; otherwise, a failure in the middle of the free function could leak memory or leave the system in an inconsistent state. To safe-guard this and enable lock-free compatibility a pre-check function, H5A__ff_precheck(), is introduced to ensure that the VOL object is still in a valid state before executing the free function. This check is a fast, read-only test that does not blindly dereference internal pointers. This pre-check performs assertion checks on possible invariants that aren't runtime errors but rather a violation of the

expected usage and therefore the program should be halted. These checks include checking whether the connector, connector->cls, and attribute close callback are all non-null. Otherwise, the reference count (rc), an atomic reference counter tracking how many references are held to the VOL object, is checked to ensure that the value is not 0. If rc == 0 here, the object has already begun to teardown and it's no longer safe to touch. If this is the case, the pre-check returns false and the free does not need to occur at a later time because the object has been freed or is already being freed. A check is also made to see if rc == 1 (currently in the final close) and on_fl == true (another thread has already added the object to the free list). In this case, the pre-check returns false because the deletion is already pending and does not need to be retried. The object can safely remain on the free list for eventual cleanup.

Below is an outline of the pre-check function:

```
bool H5A__ff_succeed(void *vol_obj_ptr)
{
  /* Check if the VOL object pointer itself is NULL.
     * This indicates an invalid or already-freed HDF5 ID.
     * In a lock-free context, we must avoid dereferencing in this case.
     */
    if (vol_obj_ptr == NULL)
        return false;

    H5VL_object_t *vol_obj = (H5VL_object_t *)vol_obj_ptr;

    // If the reference count is zero, the object is already in teardown or freed.
    // No references remain, so it cannot be freed again without error.
    #ifdef H5_HAVE_MULTITHREAD
    if (atomic_load(&(vol_obj->rc)) == 0)
        return false;
    #else
    if (vol_obj->rc == 0)
        return false;
    #endif

    // If this is the final reference (rc == 1) and the object is already on a free list (on_fl == true),
    // then a free operation has already been scheduled. A second free would cause a double-
free assertion.
    #ifdef H5_HAVE_MULTITHREAD
    if ((atomic_load(&(vol_obj->rc)) == 1) && atomic_load(&(vol_obj->on_fl)))
        return false;
    #else
    if ((vol_obj->rc == 1) && vol_obj->on_fl)
        return false;
    #endif
```

```
    // At this point, it is safe to free the object.
    // (The object has references and is not already being freed.)
    // The assertions below ensure the VOL object is structurally sound.
    assert(vol_obj->connector != NULL);
    assert(vol_obj->connector->cls != NULL);
    assert(vol_obj->connector->cls->attr_cls.close != NULL);

    return true;
}
```