Threadsafe H5VL Design

Matthew Larson

July 31, 2024

Contents

1 Introduction

This document is an overview of the changes necessary (as of HDF5 1.14.2) to make the H5VL module threadsafe. Some modules which H5VL makes use of - such as H5P or H5CX - have existing threadsafe design proposals which will be referenced when relevant.

2 Global Variables

2.1 Global Connector IDs

The HDF5 library uses global variables to track the registration status and ID of the native VOL connector (`H5VL_NATIVE_ID_g`) and the Internal Passthrough VOL connector (`H5VL_PASSTHRU_g`).

Each of these global IDs is used during the respective VOL's registration and termination, and is used in a read-only manner in a handful of other places in the library. Each ID is publicly exposed for reading via a macro that returns the result of their registration callback - `H5VL_NATIVE` and `H5VL_PASSTHRU`.

It should be fairly trivial to replace each of the the writes to these global variables with atomic compare-and-swap operations to prevent conflicts during registration, and to replace all later reads with atomic reads.

Since the behavior of the exposed macros in single-threaded use cases shouldn't change, and the behavior in multi-threaded cases was previously undefined, making this exposed variable atomic should not constitute a breaking API change.

Alternatively, because each connector will generally be registered and unregistered only a single time during the lifetime of a typical application, placing their registration under a mutex would also suffice to prevent conflicts between threads with a minimal impact on performance.

2.2 Default VOL Connector

The H5VL-global `H5VL_def_conn_s`, an instance of the VOL connector property structure `H5VL_connector_prop_t`, represents the VOL connector property in use by the library's default FAPL. The default connector is set by `H5VL__set_def_conn()`, which releases any previous value before populating it with a new value from the `HDF5_VOL_CONNECTOR` environment variable.

Note that while the default connector and the default FAPL both refer to the same connector, they store distinct reference-counted instances of the connector property.

At present, the default connector is modified only during H5VL initialization and termination, both of which should be single-threaded. The only potential multi-threaded access to the variable is read-only from `H5VL__is_default_conn()`. As such, no changes are needed to make concurrent access to this structure threadsafe. The designs outlined here are potential options to future-proof the field in the event that the library architecture changes.

If an API is ever created to allow applications to change the default connector during library runtime, it will be possible for applications to produce unexpected behavior by concurrently setting the default VOL connector in multiple threads. This will be the case even if the default connector is modified to support non-read-only access after library initialization - even if the library uses synchronization primitives to ensure atomic and ordered operations on the default VOL connector, each individual thread will be unable to know which VOL connector is going to be active during an operation. As such, it should be considered invalid API usage for more than one thread to attempt to set the default connector at a time. The solutions detailed here are concerned only with maintaining a consistent view of the shared memory and preventing corruption, not necessarily providing consistent behavior from the point-of-view of an application.

2.2.1 Potential Future-Proofing: Conversion to Atomic Pointer

One potential way to enable concurrent modification of the default connector is to convert the global field to an atomic pointer to dynamically allocated memory. This memory should be allocated in `H5VL__set_def_conn()` during library initialization.

Handling of this atomic pointer should be done as follows:

- When setting a new default connector:
  - Perform necessary setup on the connector structure to be set (registration, connector info construction, speculative reference count increase)
  - Use a load-linked/store-conditional operation to replace the existing pointer with a pointer to the new connector
  - If the previous operation fails (Due to another thread modifying the variable after the load), repeat until it succeeds.
  - Free the old default connector.

  An LL/SC instruction is used instead of a read followed by a compare-and-swap in order to avoid the ABA problem.

- Freeing the existing connector during termination can be done by setting the new default connector to NULL.

- When reading from the default connector in `H5VL__is_default_conn()`: the connector ID should be queried via an atomic read.

The setting of the default connector on the default FAPL will be made atomic by the H5P redesign. Before the redesign, H5P will acquire the global mutex during its operations - see Section 8.

H5VL reads from the default connector's information buffer when performing the VOL connector property's deep copy callback. This will be made threadsafe by the constraint that connector information is constant after connector initialization.

2.2.2 Potential Future-Proofing: Use a large atomic value in default conn

The changes outlined in Section 3.2.1 would, as a side effect, make concurrent modification of the default connector threadsafe, as all operations on connector information - including ones that target the default connector structure - would occur under the mutex protecting `LargeAtomic`.

3 Connector Information

VOL connectors may use the information callbacks in the `info_cls` field of `H5VL_class_t` to define a 'connector information' buffer. The size of this buffer, if any, should be provided by the connector in `H5VL_class_t->info_cls->size`. This buffer is exclusively for the internal use of the VOL connector and is never introspected by the library. However, the library does copy and free this buffer during various operations, so it is necessary to make it safe for potential concurrent access. This can be achieved by requiring the information buffer to be constant for each VOL connector after initialization. This appears to already be an assumption of the library, but it should be made explicit to developers in documentation going forward.

The rest of this section details the current behavior of the information buffer and sketches a potential design in the event that allowing modification of the info buffer becomes a necessary.

3.1 Behavior

The following library routines free or copy connector information buffers:

- `H5CX_retrieve_state()` - Connector-initiated operation to save the API Context's information to a 'context state'. As part of this process, the connector property is directly copied - which involves incrementing the reference count of the connector class ID and copying the connector information.

- `H5F__set_vol_conn()` - Copies the connector information buffer, but is only used during single-threaded library initialization, and so handling concurrent access here is not necessary.

- `H5Pget_vol_info()` - Bypasses the usual `H5P_get()` method for copying the property. The information buffer is deep-copied, and must be freed by `H5VLfree_connector_info()`.

- The VOL Connector Property Callbacks - See Section 3.2 for a full description.

The actual copying and freeing of connector information buffers are generally done by `H5VL_copy_connector_info()` and `H5VL_free_connector_info()`. These routines attempt to user VOL connector-defined info copy and free callbacks, but fallback to `malloc() + memcpy()` and `free()` respectively if such callbacks are not defined. This imposes the constraint that the VOL-defined info copy should follow the same semantics as memory allocation (e.g. it should expect to be freed later on).

3.2 The VOL Connector Property

The information buffer, along with connector class ID, is stored on the property type `H5VL_connector_prop_t`. This property is provided to file access property lists to decide the active VOL connector for an operation. Subsequent get, set, or copy operations on the properties in those property lists will result in the connector property being copied, and subsequent delete or free operations on the properties in those property lists will result in the connector property being released. (For a full description of the property callbacks' behavior, see the H5P callback census document.)

These property callback operations are expected to provide the semantics of a reference-counted copy, where each get, set, or copy is to be followed later by a free - even though the underlying information may or may not truly be deep copied. For the specific case of a VOL connector property, the ID portion has its reference count incremented, and the connector information portion is copied with `H5VL_copy_connector_info()`.

If all library-initiated copies of FAPL properties were initiated from API routines which specify that property lists will or may be copied, then it would be possible for a connector developer to write a connector and applications which take care to not modify the information buffer whenever one of those API routines was in progress. This would allow for a non-constant information buffer to still be threadsafe.

This constraint holds in the present design of the library, as each place that appears to indirectly read from the information buffer in an unsafe manner actually has exclusive access. However, part of the multi-threading project is to allow for VFDs to support mulit-threading. Once this change occurs, several of these accesses will be data races if the active VOL connector uses a non-constant information buffer.

- `H5CX_get_vol_connector_prop()` - uses `H5P_get()`, which causes a property copy. This is only used by `H5F__set_vol_conn()` during library initialization, so this is not a problem as long as initialization is single-threaded.

- `H5F__build_actual_name()` - If the current VFD is compatible with POSIX I/O calls, then this may copy a FAPL and each of its properties during the file open process. This only applies to the single-threaded Native VOL, which has no information buffer, and so even if multiple threads are concurrently using the Native VOL (and being throttled to single-threaded behavior upon entering it) this access is trivially threadsafe.

- `H5VL__file_open_find_connector_cb()` - Copies the FAPL that was initially provided to a file open operation. This FAPL copy may only occur when the default (Native) VOL fails to open a file. As such, the Native VOL must be the one specified in the VOL connector property of the copied FAPL, and the fact that it has no information buffer makes it threadsafe.

- `H5FD__splitter_populate_config()` - Copies the default FAPL. Since this is done by a VFD which requires use of the Native VOL, there is no information buffer and the access is trivially threadsafe.

- `H5F_get_access_plist()` - Copies the default FAPL. Only used by Native VOL routines, so there is no information buffer and the access is trivially threadsafe.

- `H5P__lacc_elink_fapl_(get/set/copy)()` - The default FAPL is duplicating during this set of callbacks on the link access property `H5L_ACS_ELINK_FAPL_NAME`. Since dynamic VOL connector loading can set arbitrary connectors on the default FAPL, any copy operations on link access property lists could potentially initiate non-API FAPL property copies. Fortunately, at this time there appear to no cases where the library internally initiates a copy of a link access property list.

- `H5FD__copy_plist()` - A routine used by the splitter and subfiling VFDs to copy FAPLs. Since this is used by VFDs which require use of the Native VOL, there is no information buffer and the access is trivially threadsafe.

The present design of the library indicates that the connector information buffer was intended to be constant during operation, as the connector information buffer stored on the connector property has the `const` qualifier. While this cannot prevent the active connector from copying and modifying the information concurrently in its own operations, it is indicative of a constraint which VOL connectors should follow. The need to support multi-threaded VFDs is a new motivation to maintain this constraint in the transition to threadsafety. The requirement that the connector information be constant for VOL connectors should be published and made clear to all VOL developers.

3.2.1 Potential Future-Proofing: Use a large atomic value for info

This section details a method by which modification of the connector information buffer after initialization could potentially be supported, if substantial motivation for this change arises during the design or implementation of a multi-threaded VOL connector.

The primary challenge which this design must overcome is the fact that, even if the library were to place an internal lock on the information buffer, any writes to the info buffer would come from the VOL connector callbacks, which would be unable to acquire the library's lock. For this reason, even acquiring the global lock before copying the information buffer would not suffice. It is necessary for whatever locking mechanism the library uses to be exposed to VOL callbacks. Another problem is that the connector information is exposed to the library as an untyped buffer of variable size, preventing simple atomic operations on it.

If the LargeAtomic stucture and operations specified in Appendix E is implemented, then a large atomic structure could wrap the connector information buffer. A large atomic load followed by an atomic initialization could perform an atomic copy, creating a new connector property instance. The LargeAtomic structure would be made public in order to provide VOL connectors with the definitions for the necessary access routines.

For backwards compatibility with existing connectors, it would likely be possible to use a compilation-time macro to only replace the untyped info buffer with a LargeAtomic when the library is built for multi-threading.

This is a non-exhaustive list of the changes that would be needed to manage connector information as a LargeAtomic:

- Changing the types of any routines which expose connector information to the application or connector. This includes `info_cls` callbacks and `H5Pget_vol_info()`.
- The library must properly initialize new VOL connector property objects with `large_atomic_init()`. This would affect property callback operations that copy the property, as well as `H5CX_retrieve_state()`
- The VOL connector property callbacks (e.g. `H5VL_conn_copy()`, `H5VL_conn_free()`) must atomically copy and destroy the connector information.
- `H5VL_connector_prop_t` structure must be modified to contain the information as a LargeAtomic.
- If the change is not implemented as a conditional compilation, then any existing connectors with information callbacks would require updated `info_cls` callbacks for compatibility with the new interface.

`H5P_peek()` copies the top-level `H5VL_connector_prop_t` value via a simple top-level `memcpy()`. Since this does not directly touch the actual large atomic buffer, it should not require any changes, although care should be taken that if it returns a pointer to an information buffer, then the information buffer is only introspected through the LargeAtomic access routines.

4 Reference Counting

Major structures within H5VL - `H5VL_class_t`, `H5VL_t`, and `H5VL_object_t` - are reference counted to allow access from multiple owners. In addition to having an atomic reference count, there are several constraints that must hold for operations on these structures to be threadsafe when they are shared between multiple threads:

1. These objects must have their reference count incremented before a thread attempts to acquire them from the global index or copy them. A memory fence must exist between the increment and subsequent verify/copy operations.

   This precaution should be unnecessary: if the internal interface is being used correctly, these operations are passed an object holding a reference to the structure, and so the structure should never have its reference count drop to zero concurrently. Regardless, it is best to proactively detect and prevent this behavior.

2. Similarly, any 'ownership transfer' operations must perform the incrementing of reference count due to the new owner's reference before the corresponding decrement due to removal of the original owner's reference.

3. Any modification of the objects must occur atomically.

   - For `H5VL_class_t`, the only potential violation of this constraint is in its `size` field of its `info_cls` subclass. As part of the redesign, it will be required that this value is constant after connector initialization.
   - For `H5VL_t`, the reference count will be converted to an atomic. The stored pointer to the VOL class and the class ID are constant after initialization.
   - For `H5VL_object_t`, the pointer to the connector should be constant after creation. The reference count will be converted to an atomic variable. The `data` buffer is dependent upon multi-threaded VOL connectors to modify it in a threadsafe manner; as long as the top-level pointer is not modified, the VOL layer itself will not run into threadsafety problems.

4. After the reference count of an object is decremented as part of a free or close operation, the operation must not reference or read from the object. This constraint also applies to single-threaded programs and already holds for all library operations.

The following is a list of H5VL routines which require changes in order to uphold these constraints. Some of these routines are in modules currently planned to reside under the global mutex, and so they do not strictly require any changes. For the sake of consistency and future-proofing, and because the changes are relatively simple, it is preferred to bring them in line with the threadsafe access pattern.

4.1 Require Only Conversion to Atomics

These operations already performs operations in the correct order and only need a memory fence after the incrementing of the reference count to ensure that in a multi-threaded scenario, actual execution order matches the order of instructions as written. Use of atomic variables with a strict enough memory order will result in the compiler generating the necessary memory fences.

- `H5VL_conn_copy()`
- `H5(A/D/F/G/M/O/T)close_async()`
- `H5O_refresh_metadata()`

4.2 Require Operation Reordering

These operations need to have reference-count incrementing re-ordered to occur before operations that duplicate (directly or through IDs/handles) VOL structures. A memory fence must also be added between the reference count change and the duplicating operations. Again, this memory fence will be implicitly generated by the compiler after converting the reference counts to atomics.

- `H5VL_new_connector()`
- `H5VL_create_object_using_vol_id()`
- `H5VL__new_vol_obj()`
- `H5VL__set_def_conn()`

4.3 Require Operation Reordering and Failure Handling

These operations need to have reference-count incrementing re-ordered to occur before operations that duplicate (directly or through IDs/handles) VOL structures, but also require additional logic to undo the reference count incrementing if a failure occurs at certain times (e.g. the reference count of a connector is speculatively incremented for a new object whose creation failed).

These operations will also implicitly have memory fences added via conversion of reference counts to atomic variables.

- `H5F__set_vol_conn()`
- `H5CX_retrieve_state()`
- `H5VL_create_object()`
- `H5T_own_vol_obj()`

4.4 Structure Modifications

- `H5VL_class_t` - Reference counting of this structure is handled through H5I, so no changes are necessary.
- `H5VL_t` - The `nrefs` field must be made atomic. This would affect a handful of operations in `H5VLint.c` and `H5Oflush.c`. Because the H5O operations will reside under the global mutex, technically only the H5VL operations need to be modified, but the H5O operations should still be made atomic for consistency with the field.

  This structure is library-private, so this does not constitute an API change.
- `H5VL_object_t` - The `rc` field must be made atomic. The only operations on this field occur within `H5VLint.c` and a test using library-private resources in `tmisc.c`.

  This structure is library-private, so this does not constitute an API change.

4.5 VOL Connector Registration

The same connector may be registered and unregistered with the library multiple times during the life of a single application via repeated use of the API's `H5VL(un)register_connector()`.

At present, the registration for a VOL connector proceeds as follows:

- `H5VL__register_connector_by_(class/name/value)()` is invoked.
- `H5I_iterate()` searches the global index for an already-registered VOL connector class that matches the provided class/name/value.
- If a connector is found:
  - `H5VL__get_connector_cb()` returns the connector class's ID to the registration function.
  - The registration function increments the reference count of the connector ID, to represent that another view into the underlying VOL class object will be created and returned to the caller.
- If a registered connector is not found:
  - If attempting to register a connector by name or value, instead attempt to load a matching connector class from H5PL's cache and register it.
  - If attempting to register a connector by class, directly register the provided connector class.

This process has a region of time between when the registered VOL connector is retrieved and the point at which its reference count is incremented, during which another thread could free the connector.

To eliminate this region, the use of `H5I_inc_ref()` should be removed from `H5VL__register_connector_by_(class/name/value)()` and added to `H5VL__get_connector_cb()`. This follows the pattern mentioned earlier of incrementing the reference count before attempting to acquire the object.

This change will have a few side effects:

1. Other routines in the library that use `H5VL__get_connector_cb()` without creating an `H5VL_class_t` object that will later be closed must decrement the reference count directly, to ensure that the VOL class is eventually freed. The affected routines are:

   - `H5VL__peek_connector_id_by_(name/value)()`
   - `H5VL__is_connector_registered_by_(name/value)()`

2. The `H5VL__get_connector_id_by_(name/value)()` routines currently increment the reference count of the connector returned from `H5VL__peek_connector_id_by_(name/value)()`, which is a wrapper around `H5I_iterate()` and `H5VL__get_connector_cb()`.

   This has a dangerous region between the object's acquisition and its reference count being incremented, just as with VOL registration. If the get connector callback is changed to increment the reference callback, and the peek functions have their semantics preserved by decrementing the reference count, then there still exists a dangerous region between the peek's decrementing of the reference count and the caller's incrementing of the reference count.

   To eliminate this dangerous region, the use of `H5VL__peek_connector_id_by_(value/name)()` should be replaced with a direct invocation of `H5I_iterate()` and `H5VL__get_connector_cb()` in these routines.

Once a threadsafe version of H5I is implemented, the replacement function for `H5I_iterate()` should be used - most likely some form of `H5I_get_next()`.

It is also required that H5PL acquire the global mutex upon entry, since it deals with non-threadsafe global structures. See Appendix B for more information on the internals of this module.

4.5.1 File Open

The H5VL routine that invokes the VOL connector's file open callback may perform VOL connector registration if the initial file open attempt fails.

It searches through the plugin cache for a cached connector class to register with the library and use to open the file. However, the VOL class it actually uses to open the file is not the newly registered instance, but is instead the non-reference-counted connector class stored in the plugin cache. While this is not currently an issue, since the plugin cache should only be emptied at library close, this still warrants changing.

This can be resolved by the following set of changes:

- Remove the `cls` field from `H5VL_file_open_find_connector_t`. It is redundant since the desired information can be found by unwrapping the connector ID under the connector property. Also remove the write to this field from `H5VL__file_open_find_connector_cb()`.

- Modify `H5VL__file_open()` to use `H5I_object()` to unwrap the new VOL class object provided by `H5VL__file_open_find_connector_cb()` via the connector property's ID field.

  Then, use this VOL class in the subsequent `H5VL__file_open()` call.

This is the only place in the library where `H5VL__file_open_find_connector_cb()` and `H5VL_file_open_find_connector_t` used, so no other considerations for their behavior are necessary.

4.5.2 VOL Connector Unregistration

The unregistration process requires no changes within H5VL at present, but does depend on a thread-safe H5I.

VOL connector unregistration may occur as a result of the following operations:

- `H5VLunregister_connector()` or `H5VLclose()` invocation by application
- The freeing/deletion of a VOL connector property on a FAPL (this results in the use of `H5VL_conn_free()`).
- During failure cleanup in `H5VL__set_def_conn()`

Note that failures during `H5VL_new_connector()` and `H5VL_create_object_using_id()` should not be able to result in connector unregistration, since these routines receive pre-existing references to the connector ID as a parameter which should never be cleaned up during the course of the routine, even on failure.

The actual process of connector unregistration is done through `H5I_dec_ref()` or `H5I_dec_app_ref()`. If these routines would decrease the ref count of the VOL connector ID to zero, then they first invoke the VOL connector ID's ID free callback (`H5VL__free_cls()`) before removing the VOL connector ID from the global index.

There is a class of thread-safety errors that could result from concurrent unregistration as the library currently stands. Consider the following example:

1. Thread A attempts to unregister a connector, and H5I begins to invoke the free function after seeing the reference count of the connector is 1
2. Before Thread A removes the VOL class from the global index, Thread B registers the same VOL by name, value, or class.
3. In Thread B, `H5I_iterate()` and `H5VL__get_connector_cb()` find the existing connector and perform the registration by increasing its reference count from 1 to 2.
4. Thread A performs the VOL class free functions, then removes the connector from the global index.
5. Thread B is now holding an invalid ID - it has undefined memory buffers, a reference count of 2 despite only being referenced once, and does not appear in the global index.

This problem should be resolved by the threadsafe implementation of H5I. Threadsafe H5I should offer an API which upon attempting to decrement the reference count of an ID to zero, first removes it from the publicly visible index (thus 'acquiring a lock on it') before invoking the free function. This implementation would prevent Thread B from operating on the existing connector in Step 3 above - instead, the VOL connector would be registered separately and no conflicts would occur.

This behavior from H5I does not solve all potential issues with H5I's concurrent usage. For example, if a module saves direct pointers to buffers which are stored under IDs, then H5I cannot remove the module's access to those buffers before invoking the free function. Depending on the particulars, this could result in a situation where H5I frees a buffer that a module still saves a reference to. It is the responsibility of modules which use H5I to either 1. Not store direct references to buffers that are also saved under H5I (e.g. always use H5I to access underlying information with properly ordered reference count operations) or 2. If storing direct pointers to buffers registered with H5I, the buffer must be an atomically reference-counted structure, with a threadsafe implementation of the `H5I_free_t` callback, and the structure in the buffer must always be properly reference counted in the calling module.

Another case to consider is unregistration of a VOL connector in one thread, while the connector is concurrently being used by another thread. This operation is safe, as long as the application does not improperly clone the connector ID itself. If the API is being used correctly, then in order to have the ID of the connector to free it, the calling thread must have performed an operation that incremented the reference count of the connector. Due to this reference counting, the unregistration will not result in the release of internal memory, and the other thread using the connector will be able to complete its operations.

4.6 Improper API Usage

If any thread or routine ever has access to a data buffer for an ID or object without a guarantee that the reference count for that object will stay above zero, there is the possibility that the buffer will be freed out from under the thread, leading to undefined behavior. This problem is general to all IDs in all modules planned for threadsafety, as well as all manually reference-counted objects such as `H5VL_object_t` and `H5VL_t`. If this guarantee cannot be provided, it will likely be necessary to implement a 'free list' structure which holds a reference to these buffers even after cleanup routines are used, until all threads have relinquished these buffers.

A review of the H5VL module's internal usage patterns for `H5VL_class_t`, `H5VL_t`, and `H5VL_object_t` indicates that the reference count changes detailed in this section should prevent this scenario from occurring, and so a free list should not be necessary. A full review of H5I usage is outside the scope of this document, and it has not been verified that such a scenario is impossible, for example, with how the H5I ID for `H5VL_class_t` is used in generic ID operations.

Improper application API usage is another potential source of problems; if an application frees an ID being used in another thread, undefined behavior is nearly guaranteed. This should be prevented by clear documentation of how multi-threading is to be performed, as well as testing with the Virtual Lock Assertion Scheme.

5 Dynamic Optional Operations

5.1 Global Array

For a full overview of how optional VOL operations work, see Appendix D. This section will focus on the parts of the dynamic operations module directly relevant to multi-threading in H5VL.

Dynamically registered optional operations on VOL connectors are stored in a global array of pointers to skip lists, `H5VL_opt_ops_g`. `H5VL__register_opt_operation()` is used both to initialize these skip lists, and to add individual operations into each skip list. The skips lists of optional operations for each object subclass are indexed by operation name. No provision is made within the table itself for distinctions between VOL connectors; any VOL connector querying the same operation name will receive the same operation type back.

The motivation for storing dynamic operations in a global array, rather than as attachments to individual VOL connectors, seems to have been passthrough VOL connectors. The dynamic operation table being shared allows passthrough VOL connectors to look up the operations registered by lower VOL connectors, and then invoke them by providing the retrieved operation type to `H5VL<object>_optional_op()` or `H5VLoptional()`.

If the array were not global, then in order for a passthrough VOL to invoke a particular optional operation on a lower VOL connector, the operation type information would have to be passed up and down the VOL connector stack in some way. While it might be possible to achieve this in principle with the connector information property, this would require inter-VOL memory management of shared buffers to store operation names dynamically. The global array simplifies the process by only requiring passthrough VOLs to know the names of the optional operations.

5.2 Threadsafety

Since skip lists are planned for replacement with a threadsafe alternative, assuming that the new data structure provides a similar insert/search/create interface, then the only necessary change is to list creation in `H5VL__register_opt_operation()`.

During the operation of this routine, if the destination list for a new dynamic operation does not exist, it should first create the new list, and then attempt to store it in the global array through an atomic compare-and-swap operation that compares the destination slot to NULL.

If another concurrent thread already created the list in the meantime, then this compare-and-swap operation will fail. The list which this thread created may be freed, and the list in the global array may be retrieved and used.

If there is some reason to delay the replacement of skip lists, it is possible, albeit more complicated, to implement threadsafe dynamic operation registration with the existing structures by making the same compare-and-swap change just described, but also acquiring a mutex within every `H5VLdyn_ops` routine that reads from or writes to a skip list at any time other than library shutdown. Reads must acquire the mutex in the same manner as writes, since insertion into skip lists is not atomic and malformed reads would be possible even with only a single writer.

Dynamically registered VOL operations are not freed until library shutdown. During the initial multi-threading design, library shutdown is planned to always be performed by a single thread. As such, freeing the dynamic operations need not be threadsafe, and nothing in the free process needs to be changed. If compelling reasons to permit multi-threaded shutdown are discovered during implementation, this topic will need to be revisited.

6 VOL Connector Interface

Existing VOL connectors were not created with concurrency in mind, since the HDF5 library did not provide support for it. As such, the H5VL layer needs to be able to dynamically decide whether or not the acquire the lock before using VOL callbacks, depending on the active VOL connector.

To accomplish this, we can query the existing VOL capability flag `H5VL_CAP_FLAG_THREADSAFE`, which should be set if the VOL connector supports concurrency. The global mutex will then be acquired only if this flag is not set.

Connectors for which some but not all callbacks are threadsafe may set this flag as true, and then grab the global mutex in their own callbacks via `H5TSmutex_acquire()` on a callback-by-callback basis as necessary.

7 Virtual Lock Assertion Scheme

There are many regions in the library where an assumption of no concurrency can be made, and no threadsafety changes are necessary. To document these assumptions and make testing easier, we will implement virtual read-write locks that are only compiled when debug assertions are enabled. These virtual locks will raise assertion failures when assumptions about concurrency are found to be false.

This idea can be extended to detect multi-threading related improper usage of the HDF5 API (e.g. accessing an ID with a reference count of 1 for writing from two threads). Because this would be directly relevant to application developers, it would be better to expose this part of the scheme via a property list option, and throw HDF5-level descriptive errors rather than assertion failures.

The potential scope of the virtual lock assertion scheme extends well past H5VL, as it could encompass changes to any module that has assumptions of single-threading.

Known use cases for this virtual lock assertion scheme include:

- Operations within the native VOL connector should be single threaded. By conditionally compiling read-write locks into native VOL structures (e.g. `H5F_t`, `H5G_t`) multiple threads using the same objects simultaneously will result in a failure.
- Similar to the above, library-internal objects that reside above the native VOL but not in thread-safe modules should not be operated on concurrently. This includes objects such as dataspaces (`H5S_t`) and non-committed datatypes (`H5T_t`).
- Detecting improper usage of object IDs:
  - A virtual read-write lock on H5I IDs could detect cases where the identifier is modified or freed while another thread has a reference to it.
  - Proper reference count semantics could be enforced with a semaphore on the ID. A semaphore with a count equal to the current reference count of the ID would be able to detect situations where the ID is used by at least one thread that does not own the reference. Given that providing a non-reference counted ID shouldn't cause problems for threads that only read from the ID, and that this could be considered a breaking change in what the API permits, a dedicated read-write lock on the ID or underlying object is likely a more elegant solution.

It should not be necessary to add virtual locks to the API Context, since it is already a thread-local structure.

7.1 Limits of the Virtual Locks

The virtual lock assertion scheme would not be able to detect all possible thread-related misuse of the API.

For example, a user application might repeatedly provide the same block of memory to the library for registration with H5I multiple times, from distinct pointers. The only way to detect this would be to iterate over all existing IDs at registration time and compare the addresses of memory that are pointed to. This is not a problem in cases where the library deep copies the buffer provided by the user.

Similarly, virtual locks would not be able to detect e.g. providing the same buffer for reading into in one thread and writing from in another thread.

7.2 Implementation Outline

In order to distinguish between reader threads and writer threads, the virtual lock structure added to existing library structures must consist of at least two atomic counts: one for currently active reader threads, and one for currently active writer threads.

In keeping with the pattern established by the `FUNC_ENTER_*` set of macros, the virtual locks will likely be acquired at function entry by a macro and released at function exit by a corresponding macro. The following macro types will be necessary:

- `VIRTUAL_LOCK_READER_ENTER`
- `VIRTUAL_LOCK_READER_EXIT`
- `VIRTUAL_LOCK_WRITER_ENTER`
- `VIRTUAL_LOCK_READER_EXIT`

The 'writer' macros will be used in any function that may potentially modify a provided virtually-locked structure.

7.2.1 Virtual Lock Error Class

In order for the virtual lock assertion scheme to raise its own errors on threadsafety violations that result from bad API usage, a new minor error code `H5E_LOCKVIOLATION` should be added to `H5Epubgen.h`. An error with this minor code should be raised by the virtual lock macros when a reader and writer thread attempt to control the same resource at the same time.

The major error code used could either be a new `H5E_THREADING` code, to provide for potential additional minor error codes to be added in the future, or `H5E_INTERNAL`.

8 Dependencies

Use of HDF5 free lists to optimize memory allocation operations should be disabled by defining `H5_NO_FREE_LISTS`.

Skip lists (H5SL) should be replaced with another data structure that is threadsafe and, ideally, more performant. H5VL only uses skip lists in order to manage dynamic operations.

8.1 Limitations of the Global Mutex

Modules that are not threadsafe are planned to be placed under a 'global mutex' which prevents concurrent execution from threads in those modules. Only a single thread will be able to execute in those modules at a time. However, threads in threadsafe modules may continue operating concurrently with a thread under the global lock. Due to this fact, the global mutex does not necessarily suffice to prevent all threadsafety issues. Consider the following example:

1. Module A is threadsafe, and Module B is not. Module B operations occur under a global mutex.
2. Module A invokes a Module B operation that deep copies from a buffer owned by Module A.

3. During the Module B deep copy, another thread in Module A concurrently writes to the buffer which is being deep copied.

In order for the global mutex to suffice, it must be the case that the operations in non-threadsafe modules do not read from buffers which threadsafe modules may concurrently write to, and do not write to buffers which threadsafe modules may concurrently read from.

If the buffer in the earlier example were wrapped in a Module B-controlled object, such that all read or write operation must proceed through Module B, then the global mutex would suffice to let Module A be threadsafe while using Module B.

8.2 H5I

A threadsafe implementation of H5VL requires a threadsafe implementation of H5I; merely placing it under the global mutex would not be enough to prevent thread-safety issues. See section 4.5.2 for an example illustrating the necessity of a fully threadsafe H5I.

8.3 H5P

Acquisition of the global mutex on entry to H5P routines will suffice for a threadsafe H5VL implementation. This is because reads and writes to properties occur through the H5P routines `H5P_get()` and `H5P_set()`. When properties are read, their value should be deep copied by H5P (either via `memcpy()` or a property-specific callback). Thus, no buffers are exposed in a non-threadsafe way.

The global default VOL connector property structure, since it is exposed directly to H5VL, does need to be handled specially as described in Section 2.2, but is not directly relevant to H5P.

8.4 H5CX

Although fields operations on the API context are performed through getters and setters of the form `H5CXget_<field>()` and `H5CXset_<field>()`, and the API context itself is a threadlocal structure, placing H5CX under the global mutex alone will not guarantee threadsafety.

This is due to the VOL object wrapping object. The VOL object wrapping context is an instance of `H5VL_wrap_ctx_t`, which contains a subfield `obj_wrap_ctx` defined by each passthrough VOL connector. This wrapper is used for certain wrapping operations within the library (for details, see the H5CX threadsafety document).

Because the VOL object wrapping context may be used and modified by the VOL layer in a multi-threaded manner, it is required that any multi-threaded passthrough VOL connector which defines it should defines its VOL object wrapper callbacks to operate on it in a threadsafe manner (e.g. either using only atomic operations, read-only operations on a structure defined during initialization, use of a mutex, or some other solution).

The API Context only shallow copies the provided VOL wrap context in its setter, and provides a pointer to the original buffer in its getter. The API context provides an interface to VOL connectors to save and load the API Context state, which deep copies all of the context's fields to a state object (`H5CX_state_t`). The VOL wrap context is deep copied by incrementing its internal reference count. At state free time, it is freed by decrementing its reference count.

Due to the API Context's shallow copying to the VOL wrap context, it is possible for a shared `H5VL_wrap_ctx_t` instance to be accessed and modified concurrently. This structure is not exposed outside `H5VLint.c`, so it should suffice to convert its reference count to an atomic variable, and convert modifications to it to fetch-and-add operations. The other fields on the VOL wrap context - a pointer to the connector-defined wrap context and a pointer to the owning `H5VL_t` - should be constant over the lifetime of the wrap context and safe for multi-threading.

8.5 H5E

H5E is currently not threadsafe itself due to an API which allows the error stacks of other threads to be directly modified by applications. However, H5VL's usage of H5E never provides it with access to any shared buffers, and never makes use of the problematic API calls. Thus, once H5E is placed under the global mutex, there are only two points to consider:

1. The macros `HGOTO_ERROR`, `HERROR`, and `HDONE_ERROR` as used from H5VL must not be provided with buffers that are shared between threads for their error messages. This constraint already holds for all H5VL routines.

   `H5VLunregister_opt_operation()` references a user-provided buffer within an error macro, but it must be provided as a constant buffer, so modifying it from the application concurrently is an API violation.

2. H5E allows for arbitrary user-defined callbacks to be executed when iterating through an error stack. If a user does define such a callback, it must not reference any shared state that may be accessed outside H5E.

8.6 H5SL

Placing H5SL under the global mutex will suffice for threadsafe operations until a threadsafe replacment is implemented.

While `H5SL_insert()` and `H5SL_destroy()` temporarily leave the skip list in an inconsistent state during their operation, the fact that the skip list itself is not exposed except through H5SL routines (e.g. `H5SL_count(), H5SL_search()`) means that the global mutex suffices to prevent this behavior from interfering with concurrency in H5VL.

8.7 Other Dependencies

Most modules are not planned for a threadsafe implementation and will need to acquire the global mutex on entry.

The modules H5VL directly uses which are not planned for threadsafety are:

- H5T - H5VL uses the following H5T routines:

  – `H5T_construct_datatype()` is invoked on a newly constructed VOL object that is not yet public and cannot be exposed to other threads. As such, all operations conducted within this routine should be threadsafe.

  – `H5T_get_named_type()` is a wrapper around retrieving the VOL object pointer from an H5T_t.

  – `H5T_already_vol_managed()` - Compares the `vol_obj` field of an `H5T_t` to NULL.

  These invocations are all safe, as long as modifications to `H5T_t` instances by the library are always conducted under a global mutex. If they were not, then it is possible (albeit unlikely) for a concurrent write to an `H5T_t` to be interrupted, and for one of these reads to find a malformed partially-written `vol_obj` pointer under `H5T_t`.

- H5PL - Placing this under the global mutex will suffice for a threadsafe H5VL. See Appendix B for details.

- H5FL - Should be disabled during configuration and trivially map to system memory allocation routines, which are threadsafe.

- H5MM - This module is a wrapper around system memory allocation routines, which are threadsafe.

Several more modules which are not planned for threadsafety are used by H5VL only during library initialization, which (at least during the initial implementation of multi-threading) will be carried out in a single-threaded manner. As such, these use of these modules during initialization does not technically impose any additional constraints. These modules are:

- H5T
- H5O
- H5D

- H5F
- H5G
- H5A
- H5M
- H5CX
- H5ES
- H5Z
- H5R

Any module not specifically planned for threadsafety is also assumed to be under the global mutex.

The acquisition of the global lock by routines in these modules will most likely be done through a set of `FUNC_ENTER_*` macros. This method of implementation allows for routines within a module to be considered threadsafe or not threadsafe on an individual basis.

9 Miscellaneous

9.1 Issues with single-threaded reference counting

9.1.1 H5VL_t Free with Zero Ref Count

The constructor for `H5VL_t` initializes its `nrefs` field to zero, before any VOL objects that reference the connector are instantiated. If a failure occurs before such a VOL object is created, then the cleanup routine which decrements the reference count of `H5VL_t`, `H5VL_conn_dec_rc()`, will make its reference count negative (since it is stored as a signed integer). This will prevent the object's memory from being freed.

This issue can potentially occur during `H5ESinsert_request()` and `H5VL_register_using_vol_id`.

This should be resolved by separating the cleanup of a connector instance into separate reference count manipulation and cleanup routines, and using the direct cleanup routines in cases where the connector is known to have no references.

10 Testing

10.1 Framework

In general, completely exhaustive testing of multi-threaded systems is nearly impossible due to the myriad ways thread scheduling and instruction reordering can vary across compilers, architectures, machines, and individual testing runs. As such, this test framework aims to maximize the chance of bug discovery by using as much of the VOL interface as possible from a varying number of threads with a variety of VOL connector stack setups.

Each test scenario outlined in this section should be conducted with a number of concurrent threads ranging from one to the upper bound supported by the current machine. If this makes testing take a prohibitively long time, then a random selection of threadcounts in this range can be selected for testing. Testing some pattern of fixed thread counts (e.g. power-of-2) is less likely to elicit unusual thread scheduling issues.

The test scenarios should also be conducted with each of the following VOL connector stack permutations:

- A single-threaded terminal connector
- A (no-op) multi-threaded terminal connector
- A single-threaded passthrough connector and a single-threaded terminal connector
- A single-threaded passthrough connector and a (no-op) multi-threaded terminal connector

- A multi-threaded passthrough connector and a single-threaded terminal connector
- A multi-threaded passthrough connector and a (no-op) multi-threaded terminal connector
- Two passthrough connectors, both single-threaded, and a single-threaded terminal connector
- Two passthrough connectors, the top single-threaded and the lower multi-threaded, and a single-threaded terminal connector
- Two passthrough connectors, the top multi-threaded and the lower single-threaded, and a single-threaded terminal connector
- Two passthrough connectors, both multi-threaded, and a single-threaded terminal connector

Scenarios with two passthrough connectors are included in order to test the passthrough-to-passthrough use case. It is not necessary to vary the terminal connector's threadedness in these scenarios, since all permutations of the passthrough-to-terminal connector use case are already tested.

Any single-threaded connector will implicitly throttle all connectors below it to also be single threaded due to the global mutex, but it is worth testing to make sure that this behavior is honored in each of these cases.

It is likely that the extensive nature of this testing will make it much more time and resource-intensive than the rest of the HDF5 testing suite, and so it may make sense to offer a reduced multi-threaded testing suite for users, while the main library runs the extensive testing suite at some regular interval (e.g. once per week).

10.2 VOL Connector Testing

10.2.1 API Tests

The library already has a test suite specifically for use by VOL connector: the HDF5 API tests, or 'h5 api' tests. Through the existing dynamic plugin loading system (i.e. the environment variables `HDF5_VOL_CONNECTOR` and `HDF5_PLUGIN_PATH`), it is simple to change the active VOL connector stack and re-execute this test suite.

However, due to the meaningful internal differences in H5VL between 'regular' plugin registration when manually linking static libraries and dynamic linking of shared libraries (see Section 4.5 and Appendix B), it will also be necessary to run the library's tests with manually linked VOL connector stacks. This will require modifying the library's API testing infrastructure to support running the tests with manually-linked VOL connectors by passing around a FAPL. This will entail modifying the top-level API test infrastructure in `H5_api_test.c`, adding FAPL parameters to the main object subclass routines `H5_api_<subclass>_test()`, and adding FAPL parameters to each individual API test.

It will be necessary to implement some method for the test runner to specify which manually linked VOL connector(s) to set on the FAPL and use for the API tests. This will be done through a new set of command line options for `H5_api_test.c`. It already accepts command line input, so this is an expansion of existing functionality. An option will be added to provide the names of the connectors to be registered during test execution. If connector information is required for registration of passthrough connector stacks or any other purpose, it may also be provided via another new command line option. There is precedent for providing connector information through the command line in this manner when building certain VOL connectors under the library with CMake (see here).

It would also be possible to control API tests' manually linked VOL connector stack with a new set of environment variables, similar to the existing pattern for dynamic plugins. However, the command line argument solution is preferred, since the connectors are to be registered in the opposite manner as those controlled by the currently used environment variables, and excessive usage of environment variables is generally a bad design pattern.

Because property lists are not generally modified during API operations, it should be possible to use the same FAPL for all API tests without any issues. There are some rare cases where property lists are modified from API functions (see the section on return properties in the H5P threadsafety document), but these are unlikely to pose a problem in the API tests, since these fields are purely informational and do not control API behavior.

This system, once implemented, should be used to test all VOL connector stack permutations except those that end with a no-op multi-threaded connector. This includes the default native VOL.

10.2.2 Other Tests

In addition to the API tests, the library has a suite of tests meant only to be used with the Native VOL.

Whether or not these tests should be run with a variable number of threads varies from test-to-test, based on whether or not library-internal functions are directly used.

Any test which uses library-internal functions should not be run concurrently from multiple threads, since this would be a violation of the global mutex and likely lead to errors that would never occur during actual library operation.

Any test which uses purely the public API should be run concurrently from multiple threads, in order to more thoroughly test that the global mutex allows for proper operation of the native VOL.

10.2.3 Dummy Multi-Threaded Terminal VOL

To elicit usage patterns in H5VL specific to multi-threaded VOL connectors, a dummy multi-threaded terminal VOL connector is needed. This connector should allow concurrent thread execution within VOL callbacks. Given the complexity of creating a full multi-threaded VOL connector, this testing connector should only perform no-ops and assertions.

A no-op connector won't produce the expected side-effects of VOL operations (e.g., object creation/modification), rendering the 'h5 api' tests ineffective for testing VOL connector stack permutations ending with this connector. Custom tests that do not expect these side-effects must be created. This system would invoke each VOL connector callback in isolation, with only the requirement that no errors (as detected by the virtual lock assertion scheme) occur within the library.

This dummy multi-threaded terminal VOL will be used in testing all permutations of the VOL connector stack that end with a multi-threaded connector.

10.2.4 Dummy Multi-threaded Passthrough VOL

The interest in testing passthrough VOL connectors specifically is due to the fact that they re-enter the H5VL module from the API layer during their callback operations, a pattern that may elicit issues not detected during testing of terminal connectors alone.

The existing internal passthrough connector performs no-ops before passing execution down to the terminal connector. It should be straightforward to create a multi-threaded passthrough counterpart that does the same, but with the threadsafe capability flag enabled to allow multiple concurrent threads. Since this passthrough VOL will not do any meaningful wrapping or memory work, it should be trivially threadsafe.

10.3 VOL Layer Testing

Testing the multi-threaded H5VL module should be done with the virtual lock assertion scheme enabled to detect invalid behavior.

Testing of H5VL should be able to take place largely without any VOL connectors that support multi-threaded functionality, since the issues that a multi-threaded H5VL design must resolve exist above the VOL connector's callback implementations. The exception is testing the threadsafe capability flag and the application-level mutex control, which will require a new dummy VOL.

Specific scenarios which should have dedicated tests include:

- Concurrent registration and subsequent use of the same VOL connector from multiple threads. This should work without issue as long as the application does not modify the class during registration. Specifically, the expected behavior is that an order is imposed on the registration operations and the registration which occurs second results in reference-counted usage of the first connector class.
- Concurrent registration and unregistration of the same VOL connector from different threads. Unregistrations should safely decrement the reference count of the VOL class ID, as long as the API to acquire the connector ID is used.

- Use of a VOL connector that is registered as a result of a file open failure with the native VOL (e.g. a plugin loaded and registered from the plugin cache). This will guard against any future changes in the internal memory management of H5PL. This will require construction of a test with a forced file open failure, which can likely be achieved with a built-in malformed file.
- Use of a VOL connector that uses the threadsafe capability flag, but acquires the global mutex internally in some or all of its callbacks that are not threadsafe. This primarily evaluates the library's public threadsafety module rather than H5VL. However, since the threadsafety flag for VOL connectors relies on the application-controlled mutex, it is logical to include this in the testing process.
- Concurrent registration of dynamic operations, to test that the atomic creation of lists in `H5VLdyn_ops` does not result in any additions being skipped.

10.4 Invalid API Usage Testing

We should test that specific kinds of invalid H5VL API usage are detected, at least when the virtual lock assertion scheme is enabled:

- Application-level duplication of IDs for multiple threads. This should be detected for all library IDs, not just H5VL.
- Early freeing of publicly exposed H5VL objects (`H5VL_class_t`, `H5VL_object_t`). This should be tested with early closes that immediately result in object freeing, as well as early closes on objects with reference count greater than one, such that the actual freeing occurs later.
- Given the challenge of enumerating every potential misuse of the H5VL interface, randomized tests may be beneficial. These tests should randomly execute public H5VL operations from multiple threads to trigger threading-related errors. Regular API errors and errors caught by the virtual lock assertion scheme should not cause test failures. Only bad memory accesses and undefined behavior, as detected by sanitizers, should cause the randomized tests to fail.
- Unregistration of a VOL connector in use in another thread, from a thread that duplicated the provided ID without incrementing the reference count

10.5 Systems Which Should Not Require Unique Testing

- Unregistration of dynamic optional operations, and unregistration of dynamic optional operations while those operations are in use by other threads. Only the *registration* of dynamic operations is planned for threadsafe changes beyond global mutex usage.
- Invocation of dynamic optional operations. The only difference between these operations and regular VOL operations is their usage of a module that lies under a global mutex, so additional tests shouldn't be required.

  Similarly, 'intended-for-native' optional operations should not require custom testing, since the H5VL layer treats these operations exactly the same as it treats other callback invocations that are routed to VOL connectors.
- Concurrent Registration and unregistration of VOL connectors as dynamic plugins. This is controlled by H5PL which resides under the global mutex.

11 Implementation

Implementation of the various changes required for multi-threading within H5VL should take place in the following order. This ordering was constructed with the intent that after each discrete step the library should be completely functional, in order to divide the potentially very large set of changes necessary for a threadsafe H5VL into manageable stages.

1. The single-threaded reference counting issues specified in Section 9.1 should be fixed

2. The global dynamic VOL operation table should be initialized through atomic operations.

3. Atomic reference count handling of H5VL objects:

   - Reference count operations on H5VL structures should be re-ordered as described in Section 4, with the corresponding failure handling when necessary, and the reference counts for `H5VL_t` and `H5VL_object_t` converted to atomic variables.
   - The VOL wrap context should have its reference count made atomic (Section 8.4).

4. The VOL connector registration process should be made threadsafe as described in Section 4.5

5. The multi-threaded test framework described in Section 10 should be implemented. This step includes:

   - Changes to the HDF5 API tests to allow for usage with test runner-specified manually linked VOL connector(s).
   - A framework to run the library's tests with a variable number of threads
   - A new set of test cases meant to elicit multi-threading specific errors as described in Section 10.3
   - A set of 'dummy' VOL connector callback tests for testing no-op (multi-threaded) VOL connectors.
   - Dummy multi-threaded passthrough and terminal VOL Connectors

   Note that at this point, none of these systems will be able to be run with actual multi-threading due to H5VL still remaining under the global mutex.

6. The virtual lock assertion scheme described in Section 7 should be implemented, along with the invalid API usage tests. These are paired together, since the invalid usage API tests will serve as the main means to verify the correctness and efficacy of the virtual lock assertion scheme.

7. The dependency module H5I must be made threadsafe, and a threadsafe alternative to `H5I_iterate()` (e.g. `H5I_get_next()`) should be provided.

8. The constraint that the connector information buffer and connector information buffer size are constant after connector initialization should be uniformly expected throughout the library.

9. The global mutex should be refactored to be acquired below H5VL in non-threadsafe modules, exposing H5VL for multi-threading by applications.

A H5VL Structures

A.1 H5VL_class_t

Represents a class of VOL connector. Contains pointers to its callbacks and metadata.

```
typedef struct H5VL_class_t {
    /* Overall connector fields & callbacks */
    unsigned           version;          /**< VOL connector class struct version number */
    H5VL_class_value_t value;            /**< Value to identify connector               */
    const char        *name;             /**< Connector name (MUST be unique!)          */
    unsigned           conn_version;     /**< Version number of connector               */
    uint64_t           cap_flags;        /**< Capability flags for connector            */
    herr_t (*initialize)(hid_t vipl_id); /**< Connector initialization callback         */
    herr_t (*terminate)(void);           /**< Connector termination callback            */

    /* VOL framework */
    H5VL_info_class_t info_cls; /**< VOL info fields & callbacks  */
    H5VL_wrap_class_t wrap_cls; /**< VOL object wrap / retrieval callbacks */

    /* Data Model */
    H5VL_attr_class_t     attr_cls;     /**< Attribute (H5A*) class callbacks */
    H5VL_dataset_class_t  dataset_cls;  /**< Dataset (H5D*) class callbacks   */
    H5VL_datatype_class_t datatype_cls; /**< Datatype (H5T*) class callbacks  */
    H5VL_file_class_t     file_cls;     /**< File (H5F*) class callbacks      */
    H5VL_group_class_t    group_cls;    /**< Group (H5G*) class callbacks     */
    H5VL_link_class_t     link_cls;     /**< Link (H5L*) class callbacks      */
    H5VL_object_class_t   object_cls;   /**< Object (H5O*) class callbacks    */

    /* Infrastructure / Services */
    H5VL_introspect_class_t introspect_cls; /**< Container/conn introspection class callbacks */
    H5VL_request_class_t    request_cls;    /**< Asynchronous request class callbacks */
    H5VL_blob_class_t       blob_cls;       /**< 'Blob' class callbacks */
    H5VL_token_class_t      token_cls;      /**< VOL connector object token class callbacks */

    /* Catch-all */
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id,
                       void **req); /**< Optional callback */
} H5VL_class_t;
```

A.1.1 H5VL_class_value_t

This structure is a typedef'd integer that stores the ID that uniquely identifies each VOL connector.

A.1.2 H5VL_info_class_t

This structure holds the size of the VOL information buffer (as stored on the FAPL property `H5F_ACS_VOL_CONN_NAME`), and pointers to VOL-defined callbacks to operate on this information.

```
typedef struct H5VL_info_class_t {
    size_t size;                     /* Size of the VOL info */
    void *(*copy)(const void *info); /* Callback to create a copy of the VOL info    */
    herr_t (*cmp)(int *cmp_value, const void *info1, const void *info2);
        /* Callback to compare VOL info */
    herr_t (*free)(void *info); /* Callback to release a VOL info */
    herr_t (*to_str)(const void *info, char **str);
    /* Callback to serialize connector's info into a string */
    herr_t (*from_str)(const char *str, void  **info);
```

```
                        /* Callback to deserialize a string into connector's info */
} H5VL_info_class_t;
```

A.1.3 H5VL_wrap_class_t

This structure holds callbacks used to wrap and unwrap VOL objects by passthrough connectors from outside the usual VOL operation pipeline.

```
typedef struct H5VL_wrap_class_t {
    void *(*get_object)(const void *obj); /* Callback to retrieve underlying object */
    herr_t (*get_wrap_ctx)(const void *obj, void **wrap_ctx);
        /* Callback to retrieve the object wrapping context for the connector */
    void *(*wrap_object)(void *obj, H5I_type_t obj_type, void *wrap_ctx);
        /* Callback to wrap a library object */
    void *(*unwrap_object)(void *obj);    /* Callback to unwrap a library object */
    herr_t (*free_wrap_ctx)(void *wrap_ctx);
        /* Callback to release the object wrapping context for the connector */
} H5VL_wrap_class_t;
```

A.1.4 H5VL_attr_class_t

This structure holds pointers to VOL-defined callbacks for operations on attributes.

```
typedef struct H5VL_attr_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *attr_name,
        hid_t type_id, hid_t space_id, hid_t acpl_id, hid_t aapl_id, hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *attr_name,
        hid_t aapl_id, hid_t dxpl_id, void **req);
    herr_t (*read)(void *attr, hid_t mem_type_id, void *buf, hid_t dxpl_id, void **req);
    herr_t (*write)(void *attr, hid_t mem_type_id, const void *buf, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_attr_get_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params,
        H5VL_attr_specific_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*close)(void *attr, hid_t dxpl_id, void **req);
} H5VL_attr_class_t;
```

A.1.5 H5VL_dataset_class_t

This structure holds pointers to VOL-defined callbacks for operations on datasets.

```
typedef struct H5VL_dataset_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *name,
        hid_t lcpl_id, hid_t type_id, hid_t space_id, hid_t dcpl_id, hid_t dapl_id,
        hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *name,
        hid_t dapl_id, hid_t dxpl_id, void **req);
    herr_t (*read)(size_t count, void *dset[], hid_t mem_type_id[], hid_t mem_space_id[],
        hid_t file_space_id[], hid_t dxpl_id, void *buf[], void **req);
    herr_t (*write)(size_t count, void *dset[], hid_t mem_type_id[],
        hid_t mem_space_id[], hid_t file_space_id[], hid_t dxpl_id, const void *buf[], void **req);
    herr_t (*get)(void *obj, H5VL_dataset_get_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, H5VL_dataset_specific_args_t *args, hid_t dxpl_id,
        void **req);
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*close)(void *dset, hid_t dxpl_id, void **req);
} H5VL_dataset_class_t;
```

A.1.6 H5VL_datatype_class_t

This structure holds pointers to VOL-defined callbacks for operations on datatypes.

```
typedef struct H5VL_datatype_class_t {
    void *(*commit)(void *obj, const H5VL_loc_params_t *loc_params, const char *name,
        hid_t type_id, hid_t lcpl_id, hid_t tcpl_id, hid_t tapl_id,
        hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params,
        const char *name, hid_t tapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_datatype_get_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, H5VL_datatype_specific_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*close)(void *dt, hid_t dxpl_id, void **req);
} H5VL_datatype_class_t;
```

A.1.7 H5VL_file_class_t

This structure holds pointers to VOL-defined callbacks for operations on files.

```
typedef struct H5VL_file_class_t {
    void *(*create)(const char *name, unsigned flags, hid_t fcpl_id, hid_t fapl_id,
        hid_t dxpl_id, void **req);
    void *(*open)(const char *name, unsigned flags, hid_t fapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_file_get_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, H5VL_file_specific_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*close)(void *file, hid_t dxpl_id, void **req);
} H5VL_file_class_t;
```

A.1.8 H5VL_group_class_t

This structure holds pointers to VOL-defined callbacks for operations on groups.

```
typedef struct H5VL_group_class_t {
    void *(*create)(void *obj, const H5VL_loc_params_t *loc_params, const char *name,
        hid_t lcpl_id, hid_t gcpl_id, hid_t gapl_id, hid_t dxpl_id, void **req);
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, const char *name,
        hid_t gapl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, H5VL_group_get_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, H5VL_group_specific_args_t *args, hid_t dxpl_id,
        void **req);
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*close)(void *grp, hid_t dxpl_id, void **req);
} H5VL_group_class_t;
```

A.1.9 H5VL_link_class_t

This structure holds pointers to VOL-defined callbacks for operations on links.

```
typedef struct H5VL_link_class_t {
    herr_t (*create)(H5VL_link_create_args_t *args, void *obj, const H5VL_loc_params_t *loc_params,
        hid_t lcpl_id, hid_t lapl_id, hid_t dxpl_id, void **req);
    herr_t (*copy)(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj,
                   const H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id,
                   hid_t dxpl_id, void **req);
    herr_t (*move)(void *src_obj, const H5VL_loc_params_t *loc_params1, void *dst_obj,
                   const H5VL_loc_params_t *loc_params2, hid_t lcpl_id, hid_t lapl_id,
                   hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_link_get_args_t *args,
    hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params,
    H5VL_link_specific_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*optional)(void *obj, const H5VL_loc_params_t *loc_params,
        H5VL_optional_args_t *args, hid_t dxpl_id, void **req);
} H5VL_link_class_t;
```

A.1.10 H5VL_object_class_t

This structure holds pointers to VOL-defined callbacks for operations on objects.

```
typedef struct H5VL_object_class_t {
    void *(*open)(void *obj, const H5VL_loc_params_t *loc_params, H5I_type_t *opened_type,
        hid_t dxpl_id, void **req);
    herr_t (*copy)(void *src_obj, const H5VL_loc_params_t *loc_params1, const char *src_name,
        void *dst_obj, const H5VL_loc_params_t *loc_params2, const char *dst_name,
        hid_t ocpypl_id, hid_t lcpl_id, hid_t dxpl_id, void **req);
    herr_t (*get)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_object_get_args_t *args,
        hid_t dxpl_id, void **req);
    herr_t (*specific)(void *obj, const H5VL_loc_params_t *loc_params,
        H5VL_object_specific_args_t *args, hid_t dxpl_id, void **req);
    herr_t (*optional)(void *obj, const H5VL_loc_params_t *loc_params, H5VL_optional_args_t *args,
                       hid_t dxpl_id, void **req);
} H5VL_object_class_t;
```

A.1.11 H5VL_introspect_class_t

This structure contains pointers to callbacks to introspect VOL connector metadata such as capability flags and optional operations.

```
typedef struct H5VL_introspect_class_t {
    herr_t (*get_conn_cls)(void *obj, H5VL_get_conn_lvl_t lvl,
        const struct H5VL_class_t **conn_cls);
    herr_t (*get_cap_flags)(const void *info, uint64_t *cap_flags);
    herr_t (*opt_query)(void *obj, H5VL_subclass_t cls, int opt_type, uint64_t *flags);
} H5VL_introspect_class_t;
```

A.1.12 H5VL_request_class_t

This structure contains pointers to callbacks used for asynchronous requests by VOL connectors.

```
typedef struct H5VL_request_class_t {
    herr_t (*wait)(void *req, uint64_t timeout, H5VL_request_status_t *status);
    herr_t (*notify)(void *req, H5VL_request_notify_t cb, void *ctx);
    herr_t (*cancel)(void *req, H5VL_request_status_t *status);
    herr_t (*specific)(void *req, H5VL_request_specific_args_t *args);
    herr_t (*optional)(void *req, H5VL_optional_args_t *args);
    herr_t (*free)(void *req);
} H5VL_request_class_t;
```

A.1.13 H5VL_blob_class_t

This structure containers pointers to callbacks used by VOL connectors for operations on 'blob' objects.

```
typedef struct H5VL_blob_class_t {
    herr_t (*put)(void *obj, const void *buf, size_t size, void *blob_id, void *ctx);
    herr_t (*get)(void *obj, const void *blob_id, void *buf, size_t size, void *ctx);
    herr_t (*specific)(void *obj, void *blob_id, H5VL_blob_specific_args_t *args);
    herr_t (*optional)(void *obj, void *blob_id, H5VL_optional_args_t *args);
} H5VL_blob_class_t;
```

A.1.14 H5VL_token_class_t

This structure contains pointers to callbacks used by VOL connectors for operations on object tokens.

```
typedef struct H5VL_token_class_t {
    herr_t (*cmp)(void *obj, const H5O_token_t *token1, const H5O_token_t *token2, int *cmp_value);
    herr_t (*to_str)(void *obj, H5I_type_t obj_type, const H5O_token_t *token, char **token_str);
    herr_t (*from_str)(void *obj, H5I_type_t obj_type, const char *token_str, H5O_token_t *token);
} H5VL_token_class_t;
```

A.2 H5VL_t

A particular instance of a VOL class defined by an `H5VL_class_t`.

```
typedef struct H5VL_t {
    const H5VL_class_t *cls;   /* Pointer to connector class struct                  */
    int64_t            nrefs; /* Number of references by objects using this struct    */
    hid_t              id;    /* Identifier for the VOL connector                     */
} H5VL_t;
```

A.3 H5VL_object_t

A wrapper structure for objects returned from the VOL layer.

```
typedef struct H5VL_object_t {
    void   *data;      /* Pointer to connector-managed data for this object    */
    H5VL_t *connector; /* Pointer to VOL connector struct                      */
    size_t  rc;        /* Reference count                                      */
} H5VL_object_t;
```

A.4 H5VL_connector_prop_t

Information about a connector to be stored on a FAPL.

```
typedef struct H5VL_connector_prop_t {
    hid_t       connector_id;   /* VOL connector's ID                                 */
    const void *connector_info; /* VOL connector info, for open callbacks             */
} H5VL_connector_prop_t;
```

B H5PL

The library's plugin module consists primarily of two global structures: a "plugin path table" of paths to search for dynamic plugins (`H5PL_paths_g`), and a cache of plugins that have been previously loaded (`H5PL_cache_g`). A dynamic plugin is a VOL connector, a virtual file driver, or an I/O filter that is loaded from a dynamically linked library (a library with the `.dll` extension on Windows, or the `.so` extension on a POSIX system) during the runtime of the HDF5 library.

B.1 Plugin Path Table

The plugin path table is a simple array of paths. When a plugin of a given type is requested, each path in this table is searched to find a dynamic plugin that fits the provided criteria. This search is carried out in the following circumstances:

- During file driver registration (`H5FD_register_driver_by_(name/value)()`), to find the provided file driver within the file system
- During VOL connector registration (`H5VL__register_connector_by_(value/name)()`)
- When searching for a valid VOL connector to open a file in `H5VL_file_open()` in the event that the native VOL connector's file open callback fails.
- Checking filter availability in `H5Z_filter_avail()`
- Loading an unloaded filter during application of the filter pipeline in `H5Z_pipeline()`

When the capacity of the path table capacity is reached, it is reallocated and extended by a fixed amount.

The plugin path search array is exposed to applications. The provided API allows for inserting into or removing from the plugin path table at any arbitrary index. This allows for the target directories to be freely modified during the runtime of an application. Note that the runtime of path insertion operations to non-tail locations on the table scales with the number of plugin paths, since each subsequent path must be shifted forward to 'make room'.

B.2 Plugin Cache

The plugin cache is a global array of `H5PL_plugin_t` objects. While it is allocated and expanded in a similar fashion to the global path table, it is not publicly exposed.

```
typedef struct H5PL_plugin_t {
    H5PL_type_t type;   /* Plugin type                          */
    H5PL_key_t  key;    /* Unique key to identify the plugin    */
    H5PL_HANDLE handle; /* Plugin handle                        */
} H5PL_plugin_t;

typedef enum H5PL_type_t {
    H5PL_TYPE_ERROR  = -1, /**< Error                */
    H5PL_TYPE_FILTER = 0,  /**< Filter               */
    H5PL_TYPE_VOL    = 1,  /**< VOL connector        */
    H5PL_TYPE_VFD    = 2,  /**< VFD                  */
    H5PL_TYPE_NONE   = 3   /**< Sentinel: This must be last!   */
} H5PL_type_t;

typedef union H5PL_key_t {
    int           id; /* I/O filters */
    H5PL_vol_key_t vol;
    H5PL_vfd_key_t vfd;
} H5PL_key_t;

#define H5PL_HANDLE             void *
```

The cache's entries consist of three elements: a `type` enum specifying the nature of this dynamic plugin, a `key` used to compare located plugins to the target plugin, and a `handle` which acts as a pointer to the plugin value itself.

During the routine to load a target plugin, `H5PL_load()`, the plugin cache is checked before beginning the search through each directory in the plugin path table. If a match is found in the cache, then the load function returns a set of plugin information defined by the plugin's "get plugin info" callback, which should return a pointer to the `H5VL_class_t` struct for VOL connector, an `H5FD_class_t` for a file driver, and an `H5Z_class2_t` for a filter. This class information is then copied by the caller before performing registration work - for example, the VOL module copies the provided class in `H5VL__register_connector()`.

B.3 Plugin Loading

The actual loading of a plugin located in one of the plugin paths is performed in `H5PL__open()`. First, `dlopen` is used to load the dynamic library by filename, before the plugin-defined introspection callbacks (`H5PL_get_plugin_type_t` and `H5PL_get_plugin_info_t`) are loaded and used to retrieve plugin-specific information. If the plugin matches the provided search criteria (filter ID, VOL class/name, VFD class/name), it is added to the plugin cache. Otherwise, the dynamic library is closed.

The motivation for the plugin cache is likely to optimize future searches by avoiding the need to open each dynamic library during a search, as well as the cost of iterating through each item in each provided directory.

B.4 Thread Safety

Since it makes heavy use of global variables, the plugin module is not threadsafe.

Acquiring a mutex on module entry to prevent concurrent access to the plugin cache and plugin path table would suffice to make this module fit into a threadsafe design.

The primary difficulty in converting this module to support multi-threading is the plugin path table, since it allows modification to any entry in the array, preventing the use of lock-free patterns for lists that can only be modified at the head or tail. Any changes to the plugin path table that allowed it to be lock-free would likely be breaking API changes.

Converting the plugin cache to be lock-free would be much simpler, since it is only ever appended to, and does not support removal of entries. This conversion would likely involve turning the plugin cache into a linked list structure with dynamically allocated individual elements.

Due to the relatively infrequent use of the plugin module during the lifetime of most applications, making H5PL multi-threaded would produce very minor gains in performance. In tandem with the difficulty of converting its global structures to lock-free structures, this suggests that placing it under a mutex is the best solution for the foreseeable future.

C Reference Count Tracking

In order to verify that the library can never change the reference count of an object held in another structure to zero, it is necessary to examine how the structs move throughout the library, and which routines perform reference count modification.

C.1 `H5VL_class_t`

C.1.1 Initialization

- `H5VLregister_connector()`, `H5VL__register_connector_by_(class/value/name)`, `H5VL__register_connector()` - These routines increment the ref count if the ID is already registered, or register the ID and set its ref count to 1. The corresponding decrement occurs in the application-invoked `H5VLunregister_connector()`. With one exception, every invocation of the internal registration functions results from an API-level registration request.
- `H5VL__file_open_find_connector_cb()` - This is the exception to the statement that all VOL connector registrations result from API-level requests. A registration via this callback may result from failing a file open with the native VOL. The resulting new connector ID is saved on the VOL connector property in the FAPL provided to the file open API call. The corresponding reference count decrement of the connector occurs in `H5Pclose()` within the VOL connector property's close callback, `H5P__facc_vol_close()`.

C.1.2 Reference Count Modification

- `H5F_get_access_plist()` and `H5P_set_vol()` set the VOL connector property on the FAPL, and increment the ref count accordingly. The reference count incrementing occurs indirectly, via `H5P_set()` $\rightarrow$ `H5VL_conn_copy()`.
- `H5F__set_vol_conn()` - Increments the reference count of the connector class when it is stored on the shared file object with a new copy of its connector information. It is decremented if a failure occurs. If it succeeds, then the corresponding decrement is in `H5F__dest()` when the shared file handle is destroyed.
- `H5CX_retrieve_state()` - increments the reference count of the connector ID, since it is stored in a new context state object. Matching decrement in `H5CX_free_state()`. The invocation of these two routines is controlled by VOL connectors.
- `H5VL_new_connector()` - Reference count of the class is incremented due to the connector being referenced from a new `H5VL_t` object. Corresponding decrement is in `H5VL_conn_dec_rc()`, when the created connector instance eventually has its `nrefs` dropped to zero.
- `H5VL_create_object_using_vol_id()` - Increments the reference count of the connector class due to constructing a new connector instance that references it. The corresponding decrement occurs when the connector is freed by `H5VL_conn_dec_rc()`, which in this case must occur when the VOL object is freed, since the connector instance is stored on no other structure.
- `H5VL__get_connector_id(by_(name/value))()` - Increases the reference count of the connector class ID due to returning it to the caller. Takes a parameter based on whether the request for the ID originated from a public API call.
  - `H5VLget_connector_id(by_(name/value))()` - Requests the ID from the API, the corresponding decrement will occur as a result of application-controlled `H5VLclose()`.
  - `H5VLunregister_connector()` - Uses this routine internally to prevent unregistration of the native VOL connector. If the reference count incrementing occurs, it is undone at the end of this routine.
  - `H5VL__set_def_conn()` - Uses this routine to retrieve the ID of an already-registered connector, which is then stored on the FAPL VOL connector property, and should later be freed at property list cleanup time.

C.2 `H5VL_t`

C.2.1 Initialization

- `H5VL_new_connector()` - Reference count of the new connector object is initialized at zero. Corresponding close routine is `H5VL_conn_dec_rc()`. `nrefs` being initialized to zero and the free routine requiring the reference count to be decremented to exactly zero lead this structure potentially never being freed, if it is cleaned up before any references are created.

- `H5VL_create_object_using_vol_id()` - Creates a new connector instance. This connector is not directly returned and is not attached to any structure besides the newly created VOL object; it is managed and freed entirely by the new VOL object that refers to it. The corresponding decrement occurs when the new VOL object is closed.

C.2.2 Reference Count Modification

- `H5(A/D/F/G/M/O/T)close_async()` - Reference count increased in preparation for possible async operation, so that the connector is not closed if the object close operation results in a file close. Reference count is decreased at the end of this routine.

- `H5VL__new_vol_obj()` - Increments the ref count of the connector instance due to a newly created VOL object referencing it. Corresponding decrement is in `H5VL_free_object()` when the referencing object is freed.

- `H5VL_create_object()` - Increments the ref count of the connector instance due to a newly created VOL object referencing it. Corresponding decrement is in `H5VL_free_object()` when the referencing object is freed.

- `H5O_refresh_metadata()` - Increments the ref count of the connector instance in order to prevent the connector from being closed due to virtual dataset refreshes. Ref count is decremented later in the same function. These modifications occur directly to the `nrefs` field and will need to be converted to atomic fetch-and-adds.

- `H5VL_set_vol_wrapper()` - Increments the reference count of connector object due to instantiating a new VOL wrap context that references the connector. The corresponding decrement occurs when the wrapping context is freed by `H5VL__free_vol_wrapper()` as invoked in `H5VL_reset_vol_wrapper()`.

C.3 `H5VL_object_t`

- `H5VL__new_vol_obj()` - VOL object is created with a reference count of 1. Corresponding destruction is in `H5VL_free_object()` when the VOL object is freed. Invokes VOL wrap callbacks on the provided object data.

- `H5VL_create_object()` - VOL object is created with a reference count of 1. Corresponding decrement is in `H5VL_free_object()` when the VOL object is freed. Does not invoke VOL wrap callbacks on the provided object data; it is stored directly on the VOL object.

- `H5VL_dataset_(read/write)()` - Instantiates a temporary VOL object with a reference count of 1. The temporary object is allocated with stack memory and automatically released at the end of this routine.

- `H5T__initiate_copy()` - Increments the reference count of the VOL object underlying a publicly exposed datatype object. This is done because the VOL object in memory is shared between the old and new datatype as a result of the datatype copy.

- `H5T_own_vol_obj()` - This routine changes the VOL object owned by a datatype object. The reference count of the old VOL object is decreases, as the datatype no longer references it, and the reference count of the new VOL object the datatype takes ownership of is increased.

D Optional Operations Overview

Dynamic optional operations are a system for allowing VOL connectors to have an arbitrary number of optional operations. Optional operations may be provided for each object subclass (e.g. file, group, datatype) or provided directly to a VOL connector.

The Dynamic Optional operations module itself, `H5VLdyn_ops`, exists only to create a mapping between application/VOL-defined operation names and integer "op type" values which uniquely identify an optional operation for an object subclass. The general pattern for intended usage within a VOL connector/application is as follows:

1. `H5VLregister_opt_operation()` is used to associate an operation name with a library-determined `op_type` value. The VOL connector saves the returned `op_type` value.
2. Later, when an optional operation is to be used, the op type value of the operation is retrieved via `H5VL_find_opt_operation()`.
3. This op type value is provided to `H5VL<object>_optional_op()` via the `args` parameter.
4. The VOL-defined optional callback should compare the provided op type value to the op type values saved in the global structure to determine which optional operation to perform.

In most cases, optional VOL operations are not directly used by any library API calls. The design philosophy seems to have been that operations other than the primary ones defined in the VOL interface (create, open, read, write, close, delete, etc.) would all fall under optional VOL operations, and should be specially registered and invoked by VOL connectors or the applications using them.

However, this pattern is broken by some special cases that seem to have been intended for use only with the native VOL connector. These "intended for native" API functions violate the patterns outlined above in the following ways:

1. Intended-for-native functions define their own op type values in their module rather than acquiring them through dynamic registration. These op type values generally have names of the form `H5VL_NATIVE_<OBJECT>_<OP>`, although some such as `H5VL_MAP_<OP>` do not include the 'NATIVE' qualifier.

   Since these values are not registered through `H5VLdyn_ops`, they cannot be found later by operation name. They are, however, public to applications.
2. Intended-for-native functions invoke the corresponding VOL-defined optional callback (if it exists) from within the library itself.
3. Intended-for-native functions are intended to operate on structures that are only guaranteed to be meaningful for the native VOL and file format, such as chunks, free sections, page buffers, the metadata cache, and file size.

The following modules make use of intended-for-native functions:

- H5Adeprec
- H5D
- H5F, H5Fdeprec, H5Fmpi
- H5Gdeprec
- H5M
- H5O, H5Odeprec
- H5Rint

While it is possible for VOL connectors to define optional callbacks for each of the intended-for-native functions, since the defined op type values are public, most VOL connectors are unable to implement most of them in a meaningful way. However, some of these functions are useful for some VOL connectors: `H5Fget_filesize()` can be useful for VOL connectors that still work within a single file or file analog, and VOL connectors such as the Async VOL still work with the native file format and can operate on native file structures like chunks. The Async VOL in particular makes use of several intended-for-native functions: `H5VL_NATIVE_DATASET_GET_NUM_CHUNKS`, `H5VL_NATIVE_DATASET_GET_CHUNK_INFO_BY_IDX`, and `H5VL_NATIVE_FILE_GET_VFD_HANDLE` among others.

E Large Atomics and Atomic Modification of Untyped Buffers

The atomic C operations provided in `stdatomic.h` only attempt to provide 'true' atomic read and write operations on types of 128 bits or less. More fundamentally, many processors only physically support truly atomic operations of up to 64 bits.

When a larger type is qualified as atomic, most compilers will still support effectively atomic operations on it via an internal locking mechanism, although the support for this varies by compiler: GCC and Clang support large, locking atomics, but MSVC does not. In the interest of minimizing dependence on compiler-specific workarounds, it would be ideal to keep all atomic operations and types below 64 bits in size.

A related issue is atomic handling of untyped buffers (e.g. a buffer pointed to by `void*`). The C11 standard does not allow the `_Atomic` qualifier to be applied to a void buffer. It is possible to use the provided synchronization primitives to allow atomic operations on such a buffer by, for example, locking the buffer during reads and writes. See Section E.1 for such an implementation.

If possible, it would be ideal to instead use atomic types and operations such that an untyped buffer can be atomically accessed, with all locking being done internally by the compiler and not the library. Unfortunately, this does not appear to be possible without maintaining multiple instances of the target buffer - see E.3 for several sketch designs and explanations of their shortcomings.

E.1 Large Atomic Operations With Explicit Lock

The following set of routines are wrappers around atomic operations that simply acquire and release a mutex before and after their operations. The `LargeAtomic` type could serve as a workaround for large atomic types on MSVC, as well as a means to atomically handle untyped buffers of variable size. Presently, there is no specific use case for this type in the multi-threaded H5VL design.

```
typedef struct {
    pthread_mutex_t mutex;
    void *value;
    size_t size;
} LargeAtomic;

/* Ownership of the 'value' buffer is transferred to the large atomic. */
void large_atomic_init(LargeAtomic *atomic, const void *value, size_t size) {
    pthread_mutex_init(&atomic->mutex, NULL);
    atomic->value = value;
}

void large_atomic_destroy(LargeAtomic* atomic) {
    pthread_mutex_lock(&atomic->mutex);
    // Prevent other threads from attempting to access the mutex afterwards
    pthread_mutex_t _mutex = atomic->mutex;
    atomic->mutex = NULL;
    pthread_mutex_destroy(&_mutex);
}

/* 'value' must be an allocated buffer with a size equal to the atomic buffer */
void large_atomic_load(LargeAtomic* atomic, void *value) {
    pthread_mutex_lock(&atomic->mutex);

    memcpy(value, atomic->value, atomic->size);

    pthread_mutex_unlock(&atomic->mutex);
    return value;
}

/* 'value' must be a buffer with a size equal to the atomic buffer */
void large_atomic_store(LargeAtomic* atomic, void* value) {
    pthread_mutex_lock(&atomic->mutex);
    memcpy(atomic->value, value, atomic->size);
    pthread_mutex_unlock(&atomic->mutex);
}

bool large_atomic_compare_exchange(LargeAtomic* atomic, void* expected, void* desired) {
    pthread_mutex_lock(&atomic->mutex);
    bool ret_value = true;

    if (!memcmp(atomic->value, expected)) {
        memcpy(atomic->value, desired, atomic->size);
        ret_value = true;
    } else {
        memcpy(expected, atomic->value, atomic->size);
        ret_value = false;
    }

    pthread_mutex_unlock(&atomic->mutex);
    return ret_value;
}
```

E.2 Atomic Buffers via List Versioning

Large atomic buffers could also be implemented via 'versioning' in a lock-free linked list. Each write would add a new entry to the end of the list instead of modifying an existing buffer. Since entries are never modified except for their 'next' pointer (which is not part of the target buffer for reads), concurrent reads are always threadsafe. Concurrent writes are ordered, ensuring a consistent data structure.

For larger buffers, allowing this list to grow without bound could cause problematically high memory usage. Imposing a max length on the list would limit potential memory usage by dropping the oldest buffers from the head of the linked list as this limit is exceeded.

```
typedef struct {
    _Atomic int logical_size;
    _Atomic lock_free_node* head;
} LargeVersionedAtomic
```

This might seem problematic in a heavily contested list. For example, if a list is limited to 5 elements and accessed by six writers and six readers, a write could quickly be succeeded by enough other writes to push it out of the list. However, this is not an issue. As long as write and read operations are ordered, each operation will yield consistent results. The fact that any given write may not be seen by a concurrent read is inherent to any atomically accessed structure.

Lock-free singly-linked lists have well-known designs that don't need to be reproduced here. The only modification needed is to track the list's overall size. This can be done by incrementing an atomic size after each write and, if the size exceeds a predefined limit, removing the head node and decrementing the size. Since lock-free lists already support node removal, no additional work is necessary.

Consider the following pseudo-code for the write and remove element functions on a size-constrained lock-free linked list.

```
atomic_buffer_versioned_write(...) {
    Insert new buffer as element at end of list
    Fetch-and-add to increment list size

    If size exceeds limit {
        Remove head element of list
    }
}
```

```
 atomic_buffer_versioned_remove_element(...) {
     Fetch-and-add to decrement list size
     Remove head element of list
}
```

`atomic_buffer_versioned_remove_element()` could be implemented as decrementing the list size either before or after removing the actual head element. Each of these implementations produces slightly different but ultimately thread-safe behavior, as long as no thread uses the list size for allocation or traversal purposes. The choice between the implementations should be driven primarily by performance considerations.

To illustrate the safety of both implementations, consider the following scenarios. Let M be the maximum size of the list, N be some number larger than M, and k be the initial logical and physical size of the list where $k < M$. To simplify the illustration, also assume that $k+1 < M$ and $M+1 < N$.

Decrementing the logical size before removing the head node can lead to the following:

- N Threads concurrently attempt to write to the list-versioned-buffer.
- M of those N threads complete their execution of `atomic_buffer_versioned_write()` through the "decrement logical size of list" step, but are pre-empted before removing the head element.
- The actual list now contains $k + M$ elements, and the logical size is underestimated as k, since each thread has undone its incrementing of the size.
- Writer Thread M + 1 enters, adds the $(k + M + 1)$-th node to the list, then sees a logical size of $k + 1$ that indicates the list is below its maximum size and does not remove a node.

  (Note that even if many threads concurrently enter at this step, it will not be possible for them to leave the list above its maximum physical size. This is because each thread increments the size earlier in its execution, so if N additional threads enter at this point and add to the list, the maximum logical size achievable is $k + N$ and the size check of at least $k + N - M$ of these writers would observe a logical size greater than M and initiate a node removal.)
- Eventually, the pre-empted writers complete their write operations and remove a total of M nodes. The list has a final physical size of $k + 1$ and a final logical size of $k + 1$

The list might temporarily exceed its maximum size, but only by an amount up to the number of concurrent threads on the machine minus the intended maximum size. Even for relatively large buffers, this should not be a prohibitively expensive amount of memory usage given the brief duration.

Removing the head element before decrementing the size may lead to the following:

- N Threads concurrently attempt to write to the list-versioned-buffer.
- M of those N threads complete their execution of `atomic_buffer_versioned_write()` through the "remove head element of list" step, but are pre-empted before decrementing the logical size of the list.
- The actual list now contains k elements, since each writer thread has added and then removed an element from the list, but the logical size is $k + M$.
- Writer Thread M + 1 enters, adds a the $(k + 1)$-th node to the list, then sees a logical size of $k + M + 1$ that indicates the list is above its maximum size and removes the head node of the list, setting the physical size to k and the logical size to $k + M$

  (Note that, for the same reason the decrement-first implementation cannot have a logical size below k, the remove-first implementation cannot have a physical size below k no matter how many threads enter at this point.)
- Eventually, the pre-empted writers complete, decrementing the list's logical size by M. and the list's logical size will agree with the actual size of k.

The list's physical size might be unnecessarily constrained to the initial size k instead of the intended max size M, resulting in greater node removal traffic. If the list is atomically initialized with a size of 1, then this will always leave at least one valid buffer for readers.

E.3 Attempted Designs for Atomic Modification of Untyped Buffers

E.3.1 Double Buffering

A sketch design for a write routine that attempts to use two buffers and atomic switching between them:

```
typedef struct {
    void* buffers[2];
    _Atomic int active_buffer;
    size_t size;
} DoubleBufferedAtomicBuffer;

void write_atomic_buffer(DoubleBufferedAtomicBuffer* atomicBuffer, const void* data, size_t size) {
    int inactive_buffer = 1 - atomic_load(&atomicBuffer->active_buffer);
    memcpy(atomicBuffer->buffers[inactive_buffer], data, size);
    atomic_store(&atomicBuffer->active_buffer, inactive_buffer);
}
```

This write routine is not threadsafe, since two concurrent uses of `write_atomic_buffer()` would lead to a race condition on `atomicBuffer->buffers[inactive_buffer]`. Preventing such a race condition requires a mutex or mutex-like atomic flag, failing to achieve the goal of no library-level locks.

E.3.2 Pointer Swaps with Internal Allocation

Suppose that atomic writes on the un-typed buffer are implemented via the use of an atomic pointer swap to a buffer that is allocated from within the 'atomic' routine:

```
typedef struct {
    _Atomic void *buffer
    size_t size;
} AtomicBuffer;

bool init_atomic_buffer(AtomicBuffer* atomicBuffer, void *buffer, size_t size) {
    bool ret_value = true;
    void *new_buffer;

    if (NULL == (new_buffer = malloc(size))) {
        ret_value = false;
    } else {
        memcpy(new_buffer, buffer, size);
        // init_atomic_buffer should not be invoked on the same object from
        multiple threads; if this cannot be guaranteed,
        then this store may be replaced with a compare-exchange loop.
        atomic_store(&atomicBuffer->buffer, new_buffer);
    }

    return ret_value;
}

bool write_atomic_buffer(AtomicBuffer* atomicBuffer, const void* data) {
    bool ret_value = true;
    void *new_buffer;
    void *old_buffer = NULL;

    if (NULL == (new_buffer = malloc(atomicBuffer->size))) {
        ret_value = false;
    } else {
        memcpy(new_buffer, data, atomicBuffer->size);
        // Atomically Swap Pointers
        do {
            old_buffer = atomic_load(&atomicBuffer->buffer);
        } while (!atomic_compare_exchange_weak(&atomicBuffer->buffer, &old_buffer,  &new_buffer));

        // Once the above swap operation succeeds, this routine must
        // have the last reference to old_buffer : free it
        (since read_atomic_variable performs a memcpy) - free old buffer
        free(old_buffer);
    }

    return ret_value;
}
```

This implementation of an atomic write makes it impossible to implement a safe atomic read. A corresponding atomic read would need to use `memcpy()` to read the value, before doing some kind of atomic check to see if the value changed concurrently with the read. There are two significant problems with this.

Suppose the following implementation for a read routine was used:

```
bool read_atomic_buffer(AtomicBuffer* atomicBuffer, void **dst) {
    bool ret_value = true;
    void *dst_buf;
    void *old_buffer = NULL;

    if (NULL == (dst_buf = malloc(atomicBuffer->size))) {
        ret_value = false;
    } else {
        do {
            atomic_load(&old_buffer, &atomicBuffer->buffer);
            memcpy(dst_buf, atomicBuffer->buffer, atomicBuffer->size);
            atomic_load(&old_buffer2, &atomicBuffer->buffer);
        } while (old_buffer != old_buffer2);
    }

    return ret_value;
}
```

The first problem is that a concurrent write could free the buffer that the read is copying from, causing it to read from unallocated memory. This is not necessarily catastrophic as long as the result is immediately discarded without being used.

The second problem is the ABA problem. Specifically, suppose that the 'atomic check' used to see if a write has occurred is a comparison of the atomic's pointer address at two different points in time. Then the following series of events is possible:

- The original atomic buffer has data stored at address 1.
- Thread A invokes `read_atomic_buffer()`, atomically loads address 1, and begins a `memcpy()`.
- Thread B invokes `write_atomic_buffer()`. It allocates new memory at address 2, claims ownership of the buffer at address 1, and frees it.
- Thread A copies invalid memory from address 1.
- Thread C invokes `write_atomic_buffer()`. Because address 1 is now free, the system allocates heap memory at address 1. Thread C then claims ownership of address 2, publishes address 1 on the atomic buffer structure, and frees address 2.

- Thread A atomically loads the address of the buffer again. It finds that the address is now address 1. Because the address was also address 1 before the `memcpy()`, it thinks that no write occurred and the read buffer is valid, even though it is invalid.

Since only the pointers themselves are atomic, and not the buffers, it is not possible to rework `read_atomic_buffer()` to safely compare buffer contents to check if a concurrent write has occurred.

Note that `write_atomic_buffer()` itself cannot encounter the ABA problem. To see why, consider what how it would: In order for it to encounter a case where a buffer allocated and stored on atomicBuffer in another thread is equal to the previously loaded `old_buffer`, because the write always allocates `new_buffer` before freeing `old_buffer`, this would require another thread to allocate the same memory region twice simultaneously, which cannot occur.

E.3.3 Pointer Swaps with External Allocation

```
typedef struct {
    _Atomic void *buffer
    size_t size;
} AtomicBuffer;

/* AtomicBuffer takes ownership of provided buffer */
bool init_atomic_buffer(AtomicBuffer *atomicBuffer, void *buffer, size_t size) {
    atomicBuffer->buffer = buffer;
    atomicBuffer->size = size;
}


void write_atomic_buffer(AtomicBuffer *atomicBuffer, void *new) {
    void *old_buf;

    old_buf = atomic_exchange(&atomicBuffer->buffer,  &new);

    if (old_buf != NULL) {
        free(old_buf);
    }
}

void read_atomic_buffer(AtomicBuffer *atomicBuffer, void **out) {
    atomic_exchange(&atomicBuffer->buffer, NULL, out);
}
```

With this implementation, an atomic copy would be performed as follows:

1. Use `read_atomic_buffer()` to gain ownership of the buffer
2. The caller uses `memcpy()` to duplicate the buffer locally
3. Use `init_atomic_buffer()` with the duplicated value to initialize a new atomic buffer
4. Use `write_atomic_buffer()` to relinquish ownership of the original buffer

While this implementation may appear satisfactory at first, it is really hiding another library-level lock. Two concurrent read attempts will result in one taking ownership of the buffer, and the other receiving the NULL buffer. If the second thread requires a non-NULL buffer to successfully complete (a very likely requirement) then it must either fail or wait until the atomic buffer is re-populated with a non-NULL buffer once the other thread relinquishes ownership via a write. This pattern is effectively a more complicated library-level mutex.