Threadsafe H5VL Design

Matthew Larson

December 19, 2024

Contents

1 Introduction **4**
1.1 Assumptions . . . 4
1.2 Overview of H5VL . . . 4
1.2.1 H5VL Files . . . 4

2 Global Variables **5**
2.1 Global Connector IDs . . . 5
2.1.1 Overview . . . 5
2.1.2 Changes for Multi-Thread Safety . . . 6
2.2 Default VOL Connector . . . 6
2.2.1 Overview . . . 6
2.2.2 Changes for Multi-Thread Safety . . . 6

3 The VOL Connector Property **6**
3.1 Overview . . . 6
3.2 Connector Information Buffer . . . 7
3.2.1 Overview . . . 7
3.2.2 Thread Safety Analysis . . . 8
3.2.3 Changes for Multi-Thread Safety . . . 8

4 The VOL Object Wrapping Context **9**
4.1 Overview . . . 9
4.1.1 Object Wrapping . . . 9
4.1.2 The Object Wrapping Context . . . 9
4.1.3 Object Wrapping Callbacks . . . 9
4.2 Changes for Multi-Threading . . . 10

5 H5VL Structures **10**
5.1 Overview . . . 10
5.2 Changes for Multi-Thread Safety . . . 10

6 VOL Connector Registration **12**
6.1 Overview . . . 12
6.2 Changes for Multi-Thread Safety . . . 12
6.3 Connector Registration on File Open Failure . . . 13
6.3.1 Overview . . . 13
6.3.2 Changes for Multi-Thread Safety . . . 13
6.4 VOL Connector Unregistration . . . 13
6.4.1 Overview . . . 13
6.4.2 Changes for Multi-Threading . . . 13

7 Dynamic Optional Operations **14**
7.1 Overview . . . 14
7.2 The Global Optional Operations Array . . . 14
7.2.1 Overview . . . 14
7.2.2 Changes for Multi-Threading . . . 14

8 New Feature: Virtual Lock Assertion Scheme **14**
8.1 Behavior Overview . . . 15
8.1.1 Application Unlocks . . . 15
8.2 Limits of the Virtual Locks . . . 15

9 Dependencies **16**
9.1 H5I . . . 16
9.2 H5P . . . 16
9.3 H5CX . . . 16
9.3.1 Overview . . . 16
9.3.2 The VOL Wrapping Context . . . 16
9.3.3 The VOL Connector Property . . . 16
9.3.4 Changes for Multi-Threading . . . 16
9.4 H5E . . . 17
9.5 H5SL . . . 17
9.6 H5FL . . . 17
9.7 Other Dependencies . . . 17

10 Lock Acquisition **18**
10.1 Overview . . . 18
10.2 Changes for Multi-Thread Safety . . . 18
10.2.1 Lock Handling . . . 19

11 Miscellaneous **20**
11.1 VOL Connector Callbacks . . . 20
11.2 Issues with single-threaded reference counting . . . 20
11.2.1 H5VL_t Free with Zero Ref Count . . . 20

12 Testing **20**
12.1 Framework . . . 20
12.2 Test Scenarios . . . 20
12.3 VOL Connector Testing . . . 21
12.3.1 API Tests . . . 21
12.3.2 Other Tests . . . 22
12.3.3 Dummy Multi-Threaded Terminal VOL . . . 22
12.3.4 Dummy Multi-threaded Passthrough VOL . . . 22
12.4 VOL Layer Testing . . . 22
12.5 Systems Which Should Not Require Unique Testing . . . 23

13 Implementation **23**
13.1 Order of Changes . . . 23
13.2 Implementation Details . . . 24
13.2.1 Multi-Thread Safe Connector Iteration . . . 24

A H5VL Structures **25**
A.1 H5VL_class_t . . . 25
A.1.1 H5VL_class_value_t . . . 25
A.1.2 H5VL_info_class_t . . . 25
A.1.3 H5VL_wrap_class_t . . . 26
A.1.4 H5VL_attr_class_t . . . 26
A.1.5 H5VL_dataset_class_t . . . 26
A.1.6 H5VL_datatype_class_t . . . 27

A.1.7 H5VL_file_class_t . . . 27
A.1.8 H5VL_group_class_t . . . 27
A.1.9 H5VL_link_class_t . . . 27
A.1.10 H5VL_object_class_t . . . 28
A.1.11 H5VL_introspect_class_t . . . 28
A.1.12 H5VL_request_class_t . . . 28
A.1.13 H5VL_blob_class_t . . . 28
A.1.14 H5VL_token_class_t . . . 29
A.2 H5VL_t . . . 29
A.3 H5VL_object_t . . . 29
A.4 H5VL_connector_prop_t . . . 29

B H5PL **30**
B.1 Plugin Path Table . . . 30
B.2 Plugin Cache . . . 30
B.3 Plugin Loading . . . 31
B.4 Thread Safety . . . 31

C Reference Count Tracking **32**
C.1 `H5VL_class_t` . . . 32
C.1.1 Initialization . . . 32
C.1.2 Reference Count Modification . . . 32
C.2 `H5VL_t` . . . 33
C.2.1 Initialization . . . 33
C.2.2 Reference Count Modification . . . 33
C.3 `H5VL_object_t` . . . 33

D Detailed Dynamic Optional Operations Overview **34**

E Large Atomics and Atomic Modification of Untyped Buffers **36**
E.1 Atomic Buffers via List Versioning . . . 36

F Limitations of the Global Mutex **36**

G Internal File Access Property List Duplication **37**

H Internal VOL Property Frees/Copies **37**

1 Introduction

This document is an overview of the necessary changes (as in HDF5 1.14.2) to make the H5VL module thread-safe, with the goal of supporting concurrent operations at and eventually below the virtual object layer. Some modules which H5VL makes use of, such as H5P or H5CX, have existing thread-safe design proposals which will be referenced when relevant.

Each major section of this document begins with an outline of the current library behavior, before describing the necessary changes to make the described sections multi-thread safe.

1.1 Assumptions

This document makes a number of assumptions about the implementation of multi-threading within the library.

- Library initialization and shutdown will be performed a single-threaded manner. This will likely not be the case in the final production version, but for an initial prototype it will suffice.
- The multi-threaded rework of H5I described in this document and implemented in this branch is in use.
- Most internal library modules will remain under the global mutex. Some (H5P, H5CX) will eventually have a multi-thread-safe implementation, but for now will remain under the global mutex.

1.2 Overview of H5VL

The Virtual Object Layer (VOL) portion of the HDF5 library separates the low-level implementations of operations from the HDF5 API routines that invoke them. A write to a dataset may correspond to writing a block of bytes on a filesystem, creating an object in a cloud storage bucket, or something else entirely based on the active VOL connector(s). The application-visible behavior of the library is (ideally) identical regardless of the active VOL connector, allowing the VOL layer to be entirely transparent.

A VOL connector consists primarily of a set of several callbacks implementing major operations on each major HDF5 object class - files, datasets, groups, etc. Most HDF5 API operations on objects eventually invokes one or more of these callbacks, delegating control to the VOL connector.

The VOL connector used by a given operation is determined by the File Access Property List (FAPL) provided as an argument for file open, create, and delete operations. For all other operations, the VOL connector is determined by the library objects provided as arguments. Each library object stored information about the VOL connector used to create or access it, and this information is used to decide the VOL connector used for the majority of library operations.

A 'stack' of multiple VOL connectors may be active on any given operation. There are two kinds of VOL connectors: passthrough and terminal. Passthrough VOL connectors perform some kind of operation, such as logging, before delegating control to the next VOL connector in the stack. Terminal VOL connectors exist at the bottom of the stack and actually implement the expected operation, interacting with storage or a storage-equivalent. Only one terminal connector may exist in the stack at one time.

The H5VL module exists between the high-level API and the VOL connector callbacks. It handles the creation and destruction of generic VOL objects, associates those objects with their VOL connector and with library-defined high level objects, performs the registration and unregistration of VOL connectors, and routes operations to the appropriate VOL callbacks.

1.2.1 H5VL Files

The 'H5VL module' consists of the following files:

- H5VL.c
- H5VLcallback.c

- H5VLdyn_ops.c
- H5VLint.c
- H5VLconnector.h
- H5VLpkg.h
- H5VLprivate.h
- H5VLpublic.h
- H5VLmodule.h

The following files compose the Native VOL connector and the Internal Passthrough connector, and while they are not part of the H5VL module proper, some minor changes to them are necessary for multi-thread safety:

- H5VLpassthru.c
- H5VLpassthru.h
- H5VLconnector_passthru.h
- H5VLnative.c
- H5VLnative_private.h
- H5VLnative_attr.c
- H5VLnative_blob.c
- H5VLnative_dataset.c
- H5VLnative_datatype.c
- H5VLnative_file.c
- H5VLnative_group.c
- H5VLnative_introspect.c
- H5VLnative_link.c
- H5VLnative_object.c
- H5VLnative_token.c
- H5VLnative.h

2 Global Variables

2.1 Global Connector IDs

2.1.1 Overview

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

Each of these global IDs is used during the respective VOL's registration and termination, and is read from 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`.

2.1.2 Changes for Multi-Thread Safety

Since the native connector is registered during library initialization, cannot be unregistered, and cannot have its H5I-assigned ID change over the lifetime of the library, placing its registration under a lock should ensure thread safety with a minimal impact on performance.

The passthrough connector can be registered and unregistered multiple times, and so it makes more sense to convert it to an atomic variable.

2.2 Default VOL Connector

2.2.1 Overview

The phrase 'default VOL connector' is used at various points within the library and its documentation to refer to any of the following:

- The VOL connector identified by the library's default FAPL. This is the connector used by API routines provided with the `H5P_DEFAULT` property list.
- The static connector property `H5VL_def_conn_s`, a global instance of the VOL connector property structure `H5VL_connector_prop_t`. This is set once during library initialization, released during library shutdown, and used for just a single check during file open.
- The Native VOL Connector. This is always considered 'the default VOL' by `H5VL__is_default_conn()` and `H5_DEFAULT_VOL`.

The static connector property and the VOL on the default FAPL refer to different instances of the same VOL connector, unless `H5Pset_vol()` is used to modify the VOL connector property of the default FAPL. In this case, the static connector property will not change and the two will diverge. This behavior is likely unintentional.

2.2.2 Changes for Multi-Thread Safety

Because the static VOL connector property is constant after library initialization (with the exception of tests in `vol.c`), no changes are needed to make concurrent access to this structure thread-safe.

If any future need arises which motivates allowing the static connector to be changed during the runtime of the library, that operation will likely be uncommon enough to placing the operation under the global mutex a suitable solution. Note that the static connector property does not have access to its value mediated by H5P routines, which makes it a special case from all other connector properties, and prevents access and modification of the static connector property from automatically being threadsafe as a consequence of H5P being threadsafe (either due to the global mutex or the H5P redesign).

Given the relatively few places the static connector property is used within the library, and the potential it has to introduce threadsafety problems, it may prove useful to remove it entirely.

To prevent unexpected behavior from concurrent modification of the active VOL on the default FAPL, `H5Pset_vol()` should throw an error if it is used on the default FAPL. Technically, the library could allow this behavior, but no valid use case is known of and it seems far more likely to lead only to confusion and unexpected behavior.

3 The VOL Connector Property

3.1 Overview

The VOL connector used for file open, create, and delete operations is decided by the FAPL provided to that operation. Specifically, by the VOL connector property on the provided FAPL. The VOL connector property (`H5VL_connector_prop_t`) consists of an H5I-assigned VOL connector ID and a connector-defined information buffer.

Some specific details of H5P's property handling are relevant here. Each property on a property list is associated with a set of unique callbacks: create, get, set, copy, delete, and close. The create, get, set, and copy callbacks for a property are expected to duplicate the underlying property information in way that must later be cleaned up by either the delete or close callback.

In general, property callbacks occur not only as a direct result of API-level operations on H5P, but can also be indirectly triggered by internal library operations. A list of places where internal library operations can trigger property copy callbacks on FAPLs is included in Appendix G.

For a VOL connector property specifically, duplication consists of increasing the ID's reference count and copying the connector information buffer. Correspondingly, a delete or close consists of decreasing the reference count and releasing the information buffer.

An individual analysis of each internal operation that can trigger property duplication is unnecessary for a thread safety analysis, because all access to the VOL property is routed through H5P, which either acquires the global mutex or is thread safe.

3.2 Connector Information Buffer

3.2.1 Overview

The connector information buffer is part of the VOL connector property. Each VOL connector may, through the callbacks and fields of the `info_cls` portion of the VOL connector class structure, define an expected size and structure for its information buffer.

Purpose The information buffer is primarily used by passthrough VOL connectors, which are required to define it to contain at least the H5I-assigned ID of the next connector in the connector stack and the information buffer of the next connector in the connector stack. Passthrough connectors use this information to delegate control to the next connector in the connector stack during a handful of VOL operations, most notably file create, file open, and file delete.

However, it is possible for any VOL connector, not just passthrough connectors, to introspect the information buffer within those operations and perform arbitrary actions based on the information found.

Source The connector information buffer is provided to a VOL property on a property list in one of two ways:

- When H5Pset_vol() is used to set the VOL property on a file access property list, a connector information buffer may be provided.
- During library initialization, the environment variable HDF5_VOL_CONNECTOR may optionally contain VOL connector information in addition to a connector name. The information is expected to separated from the VOL name by a space. If this variable is defined and contains information, that information is set on both the default FAPL and the static connector property.

  The information buffer on the static connector property is properly set up at library initialization and released at library termination, but is otherwise unused at present.

Handling Connector information buffers are always bundled with the connector ID as part of a connector property. As such, copying of information buffers is performed indirectly whenever a property copy, create, get, or set operation is performed. At these times, the VOL property's internal info duplication routine (`H5VL_copy_connector_info()`) delegates copying to info_cls.copy() if it is defined, or memcpy() and malloc() otherwise.

Similarly, release of connector information buffers happens indirectly whenever a VOL connector property is closed or deleted, at which point the VOL property's internal info release routine (`H5VL_free_connector_info()`) delegates the release operation to info_cls.free() if it is defined, or free() otherwise.

There are several internal library routines which may free or copy connector information buffers as an indirect result of freeing or copying a FAPL. A list of these internal operations can be found in Appendix H. An individual analysis of each is unnecessary for a thread safety analysis, because all accesses to the VOL property are routed through H5P, which either acquires the global mutex or is thread safe due to the redesign.

3.2.2 Thread Safety Analysis

The global mutex acquisition within H5P does not suffice to automatically make the connector information buffer operations thread safe because invocation of VOL property callbacks is not always routed through H5P. For example, use of the to_str and from_str callbacks is exposed through the public API.

There are two independent factors to consider: whether or not the application provides an information buffer, and whether or not the active connector expects an information buffer (i.e. whether it is a passthrough connector, has info_cls callbacks defined, or introspects the connector information buffer during any other operations).

Application does not provide an info buffer, and VOL does not expect an info buffer No connector information-related operations are performed. Trivially thread safe.

Application does not provide an info buffer, and VOL expects an info buffer Operations that expect connector information will fail in a manner unrelated to thread safety. Trivially thread safe.

Application provides an info buffer, and VOL does not expect an info buffer The provided info buffer is only handled by the generic malloc(), memcpy() and free() routines. Since the information is deep copied to a new buffer at each property copy, the underlying info buffer is never shared between two property instances.

Application provides an info buffer, and VOL expects an info buffer Thread safe if and only if the connector information copy callback performs a deep copy. Otherwise it possible that, for example, the library frees a connector information buffer that a VOL callback is reading from, or the library reads an information buffer that a VOL callback is modifying.

Consider the following scenario as an example:

1. Thread 1 and Thread 2 begin an operation within a multi-threaded VOL callback that receives a FAPL. Each receives the ID indicating a FAPL that is properly being shared between threads
2. Thread 1 uses H5Pget_vol() to retrieve the VOL connector property. Recall that this implicitly invokes the info copy callback. Thread 1 now has direct access to an information buffer.
3. Thread 2 also uses H5Pget_vol() to retrieve the VOL connector property, and then begins a read from the information buffer.
4. Thread 1 modifies its connector information buffer.

If the information copy callback does not allocate a new buffer, then a torn write may occur. A deep-copying callback prevents scenarios such as this from being unsafe, since any operation on a information buffer after H5Pget_vol() will operate on a unique, independent information buffer.

Note that even connector-level locking within information callbacks won't prevent a scenario such as this from being unsafe, if the copy callback does not allocate a new buffer.

3.2.3 Changes for Multi-Thread Safety

No changes are necessary within the library proper to make the connector information buffer multi-thread safe. It need only be documented that the connector-defined information copy callback, if defined, should perform an actual deep copy of the underlying buffer. This is not required by the property callback interface, which permits properties to reference count and share buffers to emulate deep copies.

4 The VOL Object Wrapping Context

4.1 Overview

4.1.1 Object Wrapping

Different VOL connectors within a connector stack are likely to have different representations of the same object. As a concrete example, consider using the library's Internal Passthrough connector and the Native VOL connector to open a file. The Native VOL represents a file as an `H5F_t` instance, and the Internal Passthrough represents it as an instance of `H5VL_pass_through_t`. The translation between the `H5F_t` and `H5VL_pass_through_t` is referred to as "wrapping" the object, and the reverse is "unwrapping" it.

In the majority of cases, this process of wrapping and unwrapping is performed directly by the passthrough VOL callbacks that interact with objects. In some cases, however, the library needs to create or unwrap an ID or object pointer from somewhere that does not pass through the connector callbacks. As such, it is necessary for the library to have some way to directly invoke the VOL connector stack's object wrapping and unwrapping functionality.

The VOL object wrapping context and its associated callbacks exist to support these cases. The wrapping context stores information needed to properly pass individual objects between connectors.

4.1.2 The Object Wrapping Context

There are a number of architectural parallels between the object wrapping context and the connector information buffer:

- Both are treated as untyped buffers which are never introspected by the library
- Both have callbacks defined in a section of the VOL class (`wrap_cls` and `info_cls`)
- Both are required for Passthrough connectors to properly pass on control to lower connectors in the stack and both must include, at a minimum, the H5I-assigned ID of the next connector as well as its own counterpart for the next connector in the stack, e.g. the wrapping context structure must contain the context of the next connector, and the connector information must contain the information of the next connector.

The most significant high-level difference between the connector information buffer and the object wrapping context is how the library treats copying. The connector information has its copying delegated to a connector-defined callback, with a default fallback to malloc and free. By contrast, the wrapping context has no copy callback, and is always referenced counted by the library itself through the `H5VL_wrap_ctx_t` structure.

Note that the library is agnostic as to the specific implementation of `get_wrap_ctx`. The connector may implement this routine to provide a pointer to a global wrapping context, or allocate a new context each time. Similarly, no constraints are imposed upon what may be done with the wrapping context within the callbacks - connectors are free to modify it.

Because the wrapping context is not exposed to applications, neither an application nor the library itself will ever share the wrapping context between multiple threads. The reference count exists to manage multiple clients from within a single thread.

4.1.3 Object Wrapping Callbacks

The object wrapping class provides five connector-defined callbacks: get object, wrap object, unwrap object, get wrap context, and free wrap context. Of those, only the wrap object, get wrap context, and free wrap context callbacks interact with the wrap context itself.

- `herr_t get_wrap_ctx(const void *obj, void **wrap_ctx)` - Retrieves the VOL object wrapping context for the current connector and, implicitly, all connectors below it. `obj` is an object wrapped by this VOL connector and all connectors below it. The wrap context should be dynamically allocated under `wrap_ctx` in a manner such that it can later be released by a call to `free_wrap_ctx()`. This routine should involve recursive allocation of resources for lower VOL connectors through `H5VLget_wrap_ctx()`.

- `void *wrap_object(void *obj, H5I_type_t obj_type, void *wrap_ctx)` - Perform this VOL's wrapping operation on the provided object and return it. The provided `obj` buffer should not be modified, since it was provided by another connector and its contents cannot be introspected.
- `herr_t free_wrap_ctx(void *wrap_ctx)` - Releases the VOL wrapping context `wrap_ctx`, freeing any resources that were allocated within `get_wrap_ctx()`. This routine will involve directly freeing this connector's own allocated resources before recursively freeing the resources allocated by all lower VOLs through `H5VLfree_wrap_ctx()`.

4.2 Changes for Multi-Threading

The wrapping context's reference count should be made atomic.

It should be documented that all multi-threaded VOL connectors which define a wrapping context must allocate a separate wrapping context on each invocation of `get_wrap_ctx`. This ensures thread safety by preventing a wrapping context from ever being shared between threads, similar to the connector information buffer.

5 H5VL Structures

5.1 Overview

The H5VL module defines and works with three major structures:

- VOL Connector Classes - `H5VL_class_t`. Each VOL class is a collection of callbacks, VOL capability flags, and other information that defines a VOL connector.

  VOL classes are reference-counted by H5I.

- VOL Connector Instances - `H5VL_t`. Each VOL Connector instance exists to track information pertaining to a VOL class, and is used to indirectly associate VOL objects with their connector class.

  VOL connector instances track their own reference count, and track their corresponding VOL class both by direct pointer and by H5I-assigned ID.

- VOL Objects - `H5VL_object_t`. Each VOL object represents a particular high-level HDF5 object (file, group, dataset, etc.) as represented by a particular VOL connector. When the H5VL module and other high level modules in the library receive objects from or pass objects down to the VOL layer, the objects are exchanged as VOL objects in order to abstract away the differences in object representations between different VOL connectors.

  VOL objects track their own reference count, and store a pointer to their associated VOL connector instance. They contain an untyped data buffer which is only interpreted by the active VOL connector(s).

5.2 Changes for Multi-Thread Safety

`H5VL_t` and `H5VL_object_t` must have their reference counts converted into atomic variables. `H5VL_class_t` has its reference counts handled by H5I, which already has a multi-thread safe implementation.

In addition, there tare several constraints that must hold for operations on these structures to be multi-thread-safe:

1. These objects must have their reference count incremented *before* a thread attempts to acquire them from the global index or copy them, not after the acquisition or copy succeeds. To accomplish this, a memory fence must exist between the increment and subsequent verify/copy operations.

   This precaution should ideally be unnecessary: if the internal interface is being used correctly, these operations will always be passed an object that holds a reference to the structure, and so the structure should never have its reference count drop to zero before the operation is over. 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, the entire `H5VL_class_t` instance should be considered read-only after creation. Marking the individual `size` field as a constant is a possible solution, but would introduce backwards compatibility issues with existing connectors and so is not preferred.
    - 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 is already constant after creation, since the connector that an object belongs to can never be changed. The reference count will be converted to an atomic variable. The `data` buffer is dependent upon multi-threaded VOL connectors to modify it only in a thread-safe manner, but as long as the top-level pointer is not modified, the VOL layer itself can treat it as a constant value.

4. After the reference count of an structure 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. While some of these routines are in modules that reside under the global mutex, it is still necessary to make their access and modification patterns multi-thread-safe, as the global mutex only prevent concurrent execution of two routines that both attempt to acquire it. It is still possible for an operation under the global mutex to occur concurrently with any multi-thread-safe routine in H5VL.

Functions That Need Atomic Ref Count Handling These functions already handle reference counts and H5VL structure duplication in the correct manner, and just need slight changes to modify the reference counts through atomic increment/decrement routines.

- `H5VL_conn_copy()` - This function must also properly decrement the reference count on failure. The fact that it does not do so is currently an issue even in HDF5 1.14.2.
- `H5(A/D/F/G/M/O/T)close_async()`
- `H5O_refresh_metadata()`

Functions That Need Operation Reordering These functions need to have reference-count incrementing re-ordered to occur before operations that duplicate (directly or through IDs/handles) VOL structures.

- `H5VL_new_connector()`
- `H5VL_create_object_using_vol_id()`
- `H5VL__new_vol_obj()`
- `H5VL__set_def_conn()`
- `H5VL_create_object()`

Functions That Need 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, but the creation of that object fails).

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

6 VOL Connector Registration

6.1 Overview

Before they can be used, VOL connectors must be registered with the library and assigned an ID by H5I. The same connector may be registered multiple times, in which case the previous registration is to be discovered and its reference count incremented, to avoid needless internal duplication. VOL connectors may be unregistered by applications, and are automatically unregistered during library shutdown.

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.

6.2 Changes for Multi-Thread Safety

The VOL connector registration process has a region of time between when the registered VOL connector is retrieved from H5I/H5PL and the point at which its reference count is incremented, during which time another thread could free the connector.

To eliminate this region, the process of searching for a registered connector will be reworked for the multi-threaded case. The present multi-threaded H5I implementation exposes a new interface for iterating through IDs within an index: `H5I_get_first()` and `H5I_get_next()`. The multi-threaded H5VL design will replace its use of `H5I_iterate()` with these functions, in a manner that increments the reference count before attempting any other operations using a retrieved connector.

6.3 Connector Registration on File Open Failure

6.3.1 Overview

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 and use for another attempt at opening the file. However, the VOL class it actually for this file open attempt is not the newly registered instance, but is instead the non-reference-counted connector class stored in the plugin cache. Since the plugin cache is only emptied at library shutdown, the reference to this class should always be valid, and this behavior is safe if somewhat unintuitive.

6.3.2 Changes for Multi-Thread Safety

While the use of a cached connector class is not strictly a thread-safety issue, it still warrants changing to prevent subtle issues from creeping into the library at a later date.

The following set of changes will remove the dependence on a cached plugin:

- 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 unwrapped 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.

6.4 VOL Connector Unregistration

6.4.1 Overview

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 if the library is operating properly, failures during `H5VL_new_connector()` and `H5VL_create_object_using_id()` will 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.

6.4.2 Changes for Multi-Threading

Because unregistration of connectors is handled through H5I, which already has a multi-thread safe implementation, no changes to the unregistration process are necessary.

7 Dynamic Optional Operations

7.1 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.

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.

7.2 The Global Optional Operations Array

7.2.1 Overview

Dynamically registered optional operations on VOL connectors are stored in a global array of pointers to skip lists, `H5VL_opt_ops_g`, with one skip list per object subclass (file, group, dataset, etc.). `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 connector querying a given operation name will receive the same operation 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 buffer or VOL wrap context, this would require inter-connector 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.

7.2.2 Changes for Multi-Threading

A multi-thread safe implementation of the dynamic optional operations routines would require substantial locking to control access to the global operations table. Simply converting it to an atomic array and using atomic operations to access and modify it would not ensure thread safety. Consider two concurrent threads that each retrieve the pointer to a skip list from the array atomically. The first releases the skip list's memory, and then the second thread attempts to perform an operation on the freed skip list, leading to undefined behavior.

Given the substantial amount of locking that a threadsafe dynamic optional operations implementation would require, and the extremely infrequent of the dynamic operations routines during typical library usage, it makes the most sense to simply keep all dynamic operations routines under the global mutex. This may be revisited after the initial prototype if it proves harmful to performance.

8 New Feature: Virtual Lock Assertion Scheme

To detect invalid application usage of the API as early as possible, it will be beneficial to implement a virtual lock assertion scheme to perform sanity checks on H5I-assigned IDs at the start and end of API functions. The potential scope of the virtual lock assertion scheme extends well past H5VL, as it could be applied to any module with API functions that operate on H5I-assigned IDs.

The virtual locks will require that at the start and end of each routine, the number of concurrent references to the ID from API functions is permitted by its application reference count (i.e. that the number of routines current accessing the ID is less than or equal to its public reference count). If this assumption is violated, an assertion will be raised. This will detect errors such as application-side duplication of an ID without the required change in its reference count. This pattern has the

benefit of catching bad behavior as soon as possible, preventing it from manifesting much later on in a harder-to-debug form.

To avoid negative effects on performance, the virtual locks would only be built and used when the library is built with debug assertions enabled.

8.1 Behavior Overview

The virtual locks will be implemented using a new set of FUNC_ENTER_API* and FUNC_LEAVE_API* macros, as these are already in common usage in API routines. This will also allow the virtual locks to be disabled in routines which use the *_NOINIT macro variants, as these routines may not have the H5I module initialized to perform the relevant reference count checking. The virtual lock routines will thus be broken up into a 'virtual lock enter' which increments the virtual lock count before its assertion, and a 'virtual lock exit' which decrements the virtual lock count before its assertion.

To support multiple threads concurrently updating API-reference count information on a single ID, the virtual lock information should be stored on the ID itself. To ensure thread safety, the virtual lock information should be stored specifically within the multi-threaded ID kernel, which is only ever accessed or written to atomically. This does introduce some complications to the virtual lock scheme, as it must respect H5I's "do not disturb" lock, as well as use a compare-and-swap loop to check for and avoid interleaving with concurrent modifications.

8.1.1 Application Unlocks

An additional consideration is necessary for the virtual locks to avoid false positives for API routines which decrement the application-visible reference count of an ID. Consider the following scenario:

1. A VOL class ID is visible in two threads, and has a valid application-visible ref count of 2.
2. In thread A, `H5VLunregister_connector()` is invoked on the VOL ID. At virtual lock entry in this routine, the virtual lock count is increased to 1. The function proceeds until the actual unregistration occurs, and the application-visible reference count of the VOL ID drops from 2 to 1.
3. In thread B, another API routine, such as `H5VLunregister_connector()` is invoked. The virtual lock enter routine increments the virtual lock count to 2, finds that the virtual lock count exceeds the application-visible reference count, and raises an assertion for invalid API usage.

This implementation would leave a dangerous region between when a routine such as `H5VLunregister_connector()` decrements the application reference count of an ID, and between the virtual lock exit call. Until the virtual lock exit call is made, the virtual lock count of the ID would appear to be too high.

To address this, API routines which decrement the app ref count will indicate that an "application unlock" has taken place on the ID. Virtual lock verification of ID reference counts will compare the virtual lock count to the sum of application references plus 'application unlocks'. At virtual lock exit time, the 'application unlock' count should be decremented instead of the virtual lock count, if any application unlocks have taken place. This serves to prevent the virtual lock count from being decremented both at application ref count decrement time and at function exit.

To allow other threads to detect when an application unlock has occurred, application unlock information should also be stored in the ID kernel.

8.2 Limits of the Virtual Locks

The virtual lock assertion scheme would not be able to detect all possible thread-related misuse of the H5I 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.

9 Dependencies

9.1 H5I

A thread-safe implementation of H5VL requires a thread-safe implementation of H5I. Just placing H5I operations under the global mutex would not be enough to prevent thread-safety issues. A threadsafe implementation of H5I already exists, and this design was created with it in mind.

9.2 H5P

Acquisition of the global mutex on entry to H5P routines will suffice for a thread-safe 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-thread-safe way.

9.3 H5CX

9.3.1 Overview

The API Context may be thought of as a threadlocal, operation-local collection of global variables. Internal library routines may either write to or query to determine the behavior of certain operations, similar to a property list.

Values in the API Context may be retrieved through getters of the form `H5CXget_<field>()` and modified through setters of the form `H5CXget_<field>()`.

The API context allows VOL connectors to save its values to an "API Context State" object from which they may be re-loaded at a later time, in order to support pausing and resuming API operations while keeping the API context consistent. Unlike when values are set on the API Context itself, saving the API Context's values to state object involving deep-copying the value, whether by incremeting a reference count or allocating new memory.

9.3.2 The VOL Wrapping Context

The H5CX setter and getter for the VOL wrapping context simply store and return a pointer to the original buffer, performing no copy operation and not incrementing the wrapping context's reference count. As a result, multiple clients to H5CX within the same thread may end up referring to the same VOL wrapping context.

During API Context State creation, the wrapping context is copied to the state object by incrementing its reference count and copying its pointer.

9.3.3 The VOL Connector Property

Because the API Context stores fields from property lists provided to API operations, it also stores the VOL connector property for the current API operation. It stores the connector property in an untyped buffer, without incrementing the reference count of the class ID or attempting to copy the connector information buffer. The VOL Connector Property field of the API Context allows the property value to be accessed while bypassing H5P routines.

During API Context State creation, the class ID ref count is increased, and the connector information buffer is copied by the usual process - see Section 3.2.

The API Context stores two seemingly independent references to the same underlying VOL Connector Property for a given operation. One is the VOL Connector Property field `vol_connector_prop`, and one is indirect through the stored FAPL. While this is a point of interest for an H5CX redesign, it presents no multi-threading problems itself since the API Context is threadlocal and these two fields cannot be accessed concurrently.

9.3.4 Changes for Multi-Threading

H5CX requires no changes for the H5VL multi-thread implementation. The potentially-shared resources to which it has access (the VOL connector property, the VOL wrap context) have access mediated by H5VL, H5I, and the VOL connector itself. The constraints imposed on the connector

information buffer and the VOL wrap context ensure that access through H5CX is safe, as long as API-level objects are reference counted correcly by the application.

9.4 H5E

H5E is currently not thread-safe 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 (`op_name`) within an error macro. This buffer must be treated as constant by the user, and this should be documented.

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.

H5E is planned for a thread-safe implementation, but the global mutex should suffice for an initial prototype of H5VL.

9.5 H5SL

Skip lists are not thread safe. However, only the dynamic optional operations routines within H5VL are clients to H5SL. Since the dynamic optional operation routines are already planned to stay under the global mutex, no changes to H5SL are necessary.

9.6 H5FL

The HDF5 free lists that optimize memory allocation are not multi-thread safe. When multi-threading is enabled, use of H5FL free lists should be disabled.

9.7 Other Dependencies

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

The modules H5VL directly uses which are not planned for thread-safety 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 thread-safe.
  - `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 thread-safe H5VL. See Appendix B for details.
- H5MM - This module is a wrapper around system memory allocation routines, which are thread-safe.

Several more modules which are not planned for thread-safety 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 thread-safety 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 thread-safe or not thread-safe on an individual basis.

10 Lock Acquisition

10.1 Overview

Multi-thread support to key library packages such as H5VL and H5I will not suffice to allow applications to fully utilize multi-threaded VOL operations. While H5VL will be able handle multiple threads and execute VOL operations concurrently in isolation, higher-level routines such as H5Fcreate() and H5Dread() still acquire a global mutex at entry and hold it until exit, preventing multiple threads from reaching H5VL concurrently in the first place. Since thorough multi-threaded testing of the H5VL design and VOL callbacks is necessary before moving on to other modules, some change is necessary.

10.2 Changes for Multi-Thread Safety

The planned solution is to implement multi-thread safety for select operations within the H5F, H5D, H5A, H5T, H5L, H5G, and H5O modules that mainly delegate tasks to VOL operations. These tasks include create, open, read, write, get, close, as well as 'specific' and 'optional' operations. Since these routines do little to no internal processing before invoking a VOL callback, the solution should consist of little more than removing the global mutex acquisition, enabling concurrent execution without significant overhead.

In order to support the removal of global lock acquisition from these top-level HDF5 routines, the acquisition of the lock should be 'pushed down' to wrap around all non-multithread-safe calls that these routines make before invoking H5VL. This primarily applies to routines in H5P and H5CX, as these are the most commonly used modules during operation setup. Since these modules are planned for a thread-safe implementation in the near future, this locking should be a relatively short-term solution.

The order of events for a typical VOL operations would then be

1. Application invokes API routine corresponding to a VOL operation in a multi-threaded manner, without acquiring the global mutex
2. During setup, the lock is acquired before and released after *only* those operations that are part of non-multithread-safe modules
3. The corresponding H5VL operation is invoked without the global mutex
4. If the active VOL connector supports multi-threading, the corresponding VOL callback is invoked by H5VL without the global mutex.
5. Otherwise, the global mutex is acquired before and released after the VOL callback execution

10.2.1 Lock Handling

At present, the unmodified library simply acquires the global mutex on entry to any API routine and releases it afterwards. The introduction of API routines that do not acquire the global mutex, and the need to push certain specific modules "below" the global mutex require a means to allow specific modules or internal routines to acquire and release the global mutex within a single API operation. It is critical to prevent the lack of a global mutex from "leaking" into any non-thread safe modules, even indirectly.

There are two potential approaches:

- One approach is to have thread safe routines and modules such as H5VL lock and unlock before calling any non-thread safe routines and modules. This has the advantage of requiring relatively little work, as the threadsafe portion of the library is much smaller than the non-threadsafe portion, containing a relatively small number of routines which must be guarded.

  However, this approach introduces some maintainability issues. It requires thread safe modules to be aware of the threadsafe status of every invoked routine, and routines may be non-threadsafe very indirectly - consider an H5VL-invoked helper which performs a file operation that in some circumstances duplicates a property list, which may sometimes contain a certain property whose copy callback is not threadsafe.

- Another approach is to split the private and package-level equivalents to the API-level FUNC_ENTER and FUNC_EXIT macros into variants which do and do not acquire/release the global mutex. This is in line with the current implementation of the API-level _NO_MUTEX macros. This approach also makes the thread safety of any function a purely local concern, creating less of a maintenance burden.

  A downside of this approach is that, because an API operation may use dozens or hundreds of non-threadsafe internal routines, adding the acquisition or checking of the global mutex to every one of these routines could have a substantial impact on performance.

  Within this approach, there is another choice to make about whether internal macros should acquire the mutex by default or not.

  - Acquiring the mutex by default in most internal operations, with use of no-mutex variants in thread safe modules. This would be a more secure guarantee against unexpected unsafe usage of non-threadsafe modules from thread safe modules. However, this would exacerbate potential performance problems by massively increasing the number of global mutex operations.
  - Acquiring the mutex only in specific internal operations and modules that are invoked by threadsafe modules. This approach could have thread safety issues if unexpected modules are indirectly used by threadsafe ones, but it minimizes the performance impact of global mutex operations.

For the prototype implementation, the current approach is to use a new set of internal enter/exit macros which acquire the mutex only in modules directly used by H5VL. At present, these are H5P, H5E, H5CX, H5PL, and H5SL.

11 Miscellaneous

11.1 VOL Connector Callbacks

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 invoking 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 be acquired only if this flag is not set.

Connectors for which some but not all callbacks are thread-safe 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.

11.2 Issues with single-threaded reference counting

11.2.1 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`.

In the short term, this can be resolved by special failure handling in the `H5VL_t` constructor. In the longer term, the reference count handling of `H5VL_t` should be reworked. It would be best to deal with this when integrating with changes made to the HDF5 library after version 1.14.5, since a side effect of those changes is to remove this special case.

12 Testing

12.1 Framework

The library's existing test framework, testframe, has no support for running tests in a multi-threaded manner. Support for running a test in multiple threads concurrently will be added through a new set of flags, provided to the test framework at the time a test is added. The number of threads that the flagged tests are run with will be decided by a command-line argument.

It will be necessary to integrate the existing API tests with testframe to take advantage of this multi-thread support.

It will also be necessary to expand the test framework to properly aggregate the results of tests. For example, if a tests passes in most threads but fails in one, the overall result should be reported as a failure.

12.2 Test Scenarios

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.

Two new VOL connectors will be necessary for comprehensive testing:

- A multi-threaded terminal connector that transparently replicates Native VOL operations, so that it may be tested using the existing API tests.
- A multi-threaded passthrough connector

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

- A single-threaded terminal connector
- A multi-threaded terminal connector
- A single-threaded passthrough connector and a single-threaded terminal connector
- A single-threaded passthrough connector and a multi-threaded terminal connector
- A multi-threaded passthrough connector and a single-threaded terminal connector
- A multi-threaded passthrough connector and a 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).

12.3 VOL Connector Testing

12.3.1 API Tests

The library already has a test suite specifically for use by VOL connectors: 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.

This test suite does not support testing manually linked connectors. Testing manually-linked connectors in this fashion is not planned for the following reasons:

- Manually linked connectors generally use an H5P routine defined in their own custom header files to set that connector on a FAPL. Incorporating this into the test suite is implausible.
- Aside from the use of H5PL for loading previous dynamic connectors, the registration process is largely the same regardless of how the connector is linked, so it is unlikely that any bad behavior would be unearthed by specially testing the manual linking case.

12.3.2 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.

12.3.3 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. Since creating a fully multi-threaded VOL that can handle operations within the same file would be a substantial task, each thread in this connector will perform operations on its own file. Since the testing harness will execute a sequence of HDF5 operations that are expected to be API-compliant within each individual thread, all API tests within each thread should pass.

This connector will consist almost entirely of a small wrapper around the Native VOL that acquires the global mutex.

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.

12.3.4 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 thread-safe 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 thread-safe.

12.4 VOL Layer Testing

Testing of 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 thread-safe 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 thread-safe capability flag, but acquires the global mutex internally in some or all of its callbacks that are not thread-safe. This primarily evaluates the library's public thread-safety module rather than H5VL. However, since the thread-safety 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.

12.5 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 thread-safe 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.

13 Implementation

13.1 Order of Changes

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 thread-safe H5VL into manageable stages.

1. The single-threaded reference counting issues specified in Section 11.2 should be fixed

2. The dynamic optional operations routines should be pushed under the global mutex.

3. Atomic reference count handling of H5VL objects. Specifically, reference count operations on H5VL structures should be re-ordered as described in Section 5, with the corresponding failure handling when necessary, and the reference counts for `H5VL_t` and `H5VL_object_t` converted to atomic variables.

4. The global connector IDs should be made atomic.

5. The requirement that connector information be constant after initialization should be documented, both in comments and through the `const` modifier.

6. If necessary, in preparation for supporting multi-threaded library shutdown, the default VOL connector should have its information field wrapped in one of the LargeAtomic structures described in Appendix E.

7. The VOL connector registration process should be made thread-safe as described in Section 6

8. The lock acquisition in API routines that use VOL operations should be 'pushed down' as described in Section 10.

9. The multi-threaded test framework described in Section 12 should be implemented. This step includes:

   - 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 12.4
   - 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.

10. The virtual lock assertion scheme described in Section 8 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.

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

13.2 Implementation Details

13.2.1 Multi-Thread Safe Connector Iteration

In the present design, this is done through the new `H5VL__get_registered_connector()` routine. If the library is built without multi-threading, this compiles the same `H5I_iterate()` behavior as before. If multi-threading is enabled, then it uses the new iteration routines in a loop to search for the target connector.

Because some routines which search for the ID intend to create a new reference (`H5VL__register_connector_by_(value/class/name)`) and others do not (`H5VL__peek_connector_id_by_(name/value)`), the routine to get a registered connector takes an `inc_ref` parameter to decide whether to increment the ref count of the returned connector. This avoids having to decrement the ref count after retrieval in some routines, and guarantees that if a valid ID is returned, then it should be valid until it is released by the calling thread.

`H5VL__get_registered_connector()` must restart its search of the index if the current ID it is checking is released before it can perform the comparison. To see why, consider the problems with alternative approaches:

- If it were treated as an error, then iteration could stop prematurely due to a non-targeted VOL ID being freed and not find the target VOL ID later in the index.
- Simply continuing the iteration is not possible, since the ID that would be provided to `H5I_get_next()` has been freed.

Note that not all increment/decrement failures within `H5VL__get_registered_connector()` can fail as a result of concurrent free, and so several instances will still raise an error on failure (e.g. failure to decrement a ref count when it was incremented earlier in this routine).

The new iteration routine will be used in the following places:

1. `H5VL__register_connector_by_(value/name/class)`, with `inc_ref` = true
2. `H5VL__get_connector_id_by_(value/name)`, with `inc_ref` = true
3. `H5VL__peek_connector_id_by_(name/value)()`, with `inc_ref` = false
4. `H5VL__is_connector_registered_by_(name/value)()`, with `inc_ref` = false

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() → 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 Detailed Dynamic 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. In order to operate on such a buffer in an atomic fashion, there are two options:

- Use atomic types and operations with an algorithm such that all locking is done internally by the compiler and not by the library itself. See E.1. This is the preferred solution for any operation that is repeated many times whose performance is important.
- Simply use a mutex or other synchronization primitive to lock the buffer during reads and writes. The is the preferred solution for less common operations, where performance is less important than the reduced complexity and overhead.

E.1 Atomic Buffers via List Versioning

Large atomic buffers could 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 have an order enforced upon them, ensuring a consistent data structure.

For larger buffers, allowing this list to grow without bound could cause problematically high memory usage. This can be addressed by using standard mark-for-deletion algorithms, in conjunction with either 1. reference counting entries in the list (with a reference count of zero being cause for logical deletion), or 2. deep-copying target element buffers, rendering reference counting unnecessary and allowing all elements before the tail to be logically marked for deletion.

For a reference on lock-free singly linked list implementations, see Chapter 9.8 of *The Art of Multiprocessor Programming* by Herlihy, Shavit, Luchangco, and Spear.

F Limitations of the Global Mutex

Modules that are not thread-safe 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 thread-safe 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 thread-safety issues. Consider the following example:

1. Module A is thread-safe, 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, operations in non-thread-safe modules must not read from buffers which thread-safe modules may concurrently write to, and must not write to buffers which thread-safe 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 thread-safe while using Module B. This basic pattern is used by H5P and will suffice to protect property buffers from concurrent access.

G Internal File Access Property List Duplication

This is a list of places where the library may internally duplicate properties stored on a FAPL:

- `H5CX_get_vol_connector_prop()` - uses `H5P_get()`, which causes a property copy. This is currently only used by `H5F__set_vol_conn()` during file open/creation. This is a native VOL operation, and so this copy is trivially thread-safe since the native VOL has no info buffer.
- `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 is performed during file open/creation.
- `H5VL__file_open_find_connector_cb()` - Copies the FAPL that was initially provided to a file open operation. This only occurs when the default VOL (as tracked by `H5_DEFAULT_VOL`, currently hardcoded to the 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 thread-safe.

  If future changes to the library lead to a different VOL connector with a non-empty information buffer being a valid 'default' connector, then this will still be thread-safe within the library due to the lock or redesign on H5P.
- `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 thread-safe.
- `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 thread-safe.
- `H5P__lacc_elink_fapl_(get/set/copy)()` - If the external link FAPL as specified in the property `H5L_ACS_ELINK_FAPL_NAME` is not the default FAPL, then it is duplicated by this set of callbacks.
- `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 thread-safe.

H Internal VOL Property Frees/Copies

The following internal library routines free or copy connector information buffers:

- `H5CX_retrieve_state()` - This is a 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.
- `H5CX_free_state()` - This is the release routine associated with `H5CX_retrieve_state()`.
- `H5F__set_vol_conn()` - Copies the connector information buffer as part of copying the entire VOL connector property to a target shared file object.
- The previously described VOL connector property callbacks: create, get, set, copy. As a consequence of this, any library routines that copy or free entire property lists also copy or free connector information buffers.