RFC: SWMR Requirements and Use Cases

Quincey Koziol
Elena Pourmal
Dana Robinson

This document summarizes current requirements and implementation constraints for the HDF5 Single Writer/Multiple Readers (SWMR) feature and discusses the use cases that The HDF Group proposes to address in the HDF5 Library 1.10.0 release.

Table of Contents

Introduction

The SWMR feature will allow multiple processes to concurrently access an HDF5 file. Under this scenario only one process (the Writer) is allowed to modify the HDF5 file, performing operations such as adding new datasets, groups, links and attributes, and writing data to datasets. Other processes can only read from the file (the Readers).

Implementing the full HDF5 functionality for SWMR is a challenging task that involves changes to the HDF5 file format and library, such as:

1. Extensions to the file format to indicate SWMR access and to improve efficiency in storage and access for extendible datasets.
2. Enhancements to the HDF5 library's internal data structures to work properly under SWMR.
3. Extensions to the HDF5 library's metadata cache to handle flushing metadata items in a well-defined order, to assure a consistent state of the HDF5 file for a Reader's access.
4. Improvements to the HDF5 testing framework, to verify correct behavior when multiple concurrent processes access a file.

Over the past few years, The HDF Group developers have been working on a SWMR implementation that addresses two major requirements:

- There is no communication between the Writer and the Readers.
- Access to the file is truly concurrent – neither the Writer nor the Reader locks the HDF5 file.

To assure a consistent view of the HDF5 file for the Readers, other requirements such as POSIX I/O semantics for file system access were necessary.

While the full implementation is not complete, the major enhancements in 1 – 3 above have been implemented to enable SWMR use cases for extendible datasets.

This document summarizes the current requirements and implementation constraints for SWMR. It also discusses the use cases for modifying a contiguous dataset and an extensible dataset to illustrate the challenges of the current implementation approach.

SWMR Requirements and Implementation Constraints

This section describes requirements and implementation constraints under which the SWMR feature is being implemented.

1.1 Major Requirements and Constraints

The requirements listed in this subsection affect the SWMR implementation. If requirements 1 or 2 listed below are dropped (e.g., communication between the processes is allowed), implementation of the full HDF5 functionality for SWMR becomes much easier.

1. There is no communication between the Writer and the Readers participating in the concurrent access to the HDF5 file.
2. Neither the Writer nor any Reader locks the HDF5 file.

The following constraint was added in order to address requirements 1 and 2:

3. The HDF5 file resides on a file system compliant with the POSIX I/O semantics for access[1].

1.2 Other Requirements

4. SWMR functionality requires HDF5 library version 1.10.0 and higher.
5. Files created by SWMR can be accessible by the HDF5 1.8 library.
6. The Writer and the Readers can open and close the file in an arbitrary order.
7. There is no restriction on the number of Readers.
8. Only one Writer may have the file open at a time.
9. A Reader always sees a consistent HDF5 file - no errors occur while reading HDF5 metadata and raw data (e.g., discovering group structure, traversing links, reading attributes of the HDF5 objects, discovering current sizes of the datasets, reading subsets of the datasets).
10. If the Writer crashes, it leaves the file in a non-corrupted state.
11. There is no significant I/O performance penalty for file access and modifications under SWMR.

1.3 "Light Source Community"-Specific Requirements

12. Provide a mechanism (a callback function) within the HDF5 library to inform the Writer process that changes have been propagated to the HDF5 file.
13. The Writer may use the "direct chunk write" feature to modify datasets.

[1] The POSIX I/O interface specifies that writes through the interface must be performed in a sequentially consistent manner. Writes to the file must appear as atomic operations to any readers that access the file during the write; the reader will see either all or none of any write. These semantics apply to any processes that access the file from any location. GFS, GPFS, and Lustre implement POSIX semantics, while NFS system doesn't guarantee it. For discussions, see, for example, http://etutorials.org/Linux+systems, Section 19.3 "File System Access Semantics"

1.4 Implementation constraints for the HDF5 1.10.0 release

This section spells out which operations are allowed and are not allowed in the proposed SWMR implementation.

The following operations are allowed:

1. The Writer process is allowed to modify raw data of existing datasets by:
    a. Appending data along any unlimited dimension.
    b. Modify existing data
2. The Writer process is allowed to modify the value of pre-existing attributes unless the attribute has a variable-length datatype (see 5 below).

The following operations are **not** allowed (the **corresponding HDF5 calls will fail**):

3. The Writer is **not** allowed to add any new objects to the file such as groups, datasets, links, committed datatypes and attributes.
4. The Writer is **not** allowed to delete HDF5 objects (groups, datasets, links, committed datatypes and attributes).
5. The Writer is **not** allowed to modify or append to any data items containing variably-sized datatypes (including variable-length datatypes and region references).
6. File space recycling is **not** allowed. As a result the sizes of the files modified by a SWMR Writer may be larger than files modified by non-SWMR Writers.
7. Other considerations

    It is also important to understand that SWMR operations currently have no sense of transactions or ordering, so file modifications can be propagated to the HDF5 file in ways that may be surprising for Readers. For example, suppose a Writer is appending data to a dataset and also writing summary data to an attribute attached to the dataset (perhaps a running total or average). SWMR does **not** guarantee that the attribute will always reflect the data written to the dataset, or vice-versa. Put another way, SWMR is only concerned with keeping on-disk internal HDF5 data structures coherent so API calls do not fail on partially-written files. **SWMR is currently not concerned with user-level data semantics**.

Use cases

1.5 Modifying raw data in a fixed-size dataset

Description:

Modifying raw data in a fixed-size dataset within a pre-created file and reading the modified data back.

Goal:

Read data modified by the Writer in the pre-existing fixed-size datasets in a file.

Level:

User Level

Guarantees:

- The Reader will see the changes immediately (within the imitations of the specific file system).

Preconditions:

- Readers are not allowed to modify the file.
- The datasets are not extensible.
- All datasets that are modified by the Writer exist when the Writer opens the file.

Main Success Scenario:

1. An application creates a file with the required objects (groups, datasets, and attributes).
2. The Writer opens the file and datasets in the file and modifies data in the datasets.
3. A Reader opens the file and a dataset in a file and reads the data back.
4. The Reader may see one of the following:
    a. The original data.
    b. The partially written data if the H5Dwrite call requires several POSIX write calls (which occurs when the data written are non-contiguous in the file or too large for a single POSIX write call to handle).
    c. The new data.

Discussion points:

1. In order to avoid 4.b one would need to create a copy of the original data in the file and change the library to update the dataset's header message to point at the new data when it is on the disk. Since file space recycling is not available at this implementation stage and copies of the data in the file can be very large, 4.b is unavoidable unless an application would be willing to generate a file that is much greater in size than normal.
2. The Reader will see the changes immediately.

1.6 Modifying a value of existing attribute

Goal:

Read an attribute value modified by the Writer.

Level:

User Level

Guarantees:

- Readers will see the modified attribute value after the Writer issues H5Fflush or H5Oflush call.

Preconditions:

- Readers are not allowed to modify the file.
- The attribute exists when the Reader opens the file.

Main Success Scenario:

1. An application creates a file with the required objects (groups, datasets, and attributes).
2. The Writer opens the file and an object in the file and modifies a value of an attribute on the object.
3. A Reader opens the file and the object in the file and reads the attribute back.
4. The Reader may see one of the following:
    a. The original attribute value.
    b. The new attribute value.

1.7 Appending a single chunk

Description:

Appending a single chunk of raw data to a dataset along an unlimited dimension within a pre-created file and reading the new data back.

Goal:

Read data appended by the Writer to a pre-existing dataset in a file. The dataset has one or more unlimited dimensions. The data is appended by a hyperslab that is contained in one chunk (for example, appending 2-dim planes along the slowest changing dimension in the 3-dim dataset).

Level:

User Level

Guarantees:

- Readers will see the modified dimension sizes after the Writer finishes HDF5 metadata updates and issues H5Fflush or H5Oflush calls.
- Readers will see newly appended data after the Writer finishes the flush operation.

Preconditions:

- Readers are not allowed to modify the file.
- All datasets that are modified by the Writer exist when the Writer opens the file.
- All datasets that are modified by the Writer exist when a Reader opens the file.
- Data is written by a hyperslab contained in one chunk.

Main Success Scenario:

1. An application creates a file with required objects (groups, datasets, and attributes).
2. The Writer application opens the file and datasets in the file and starts adding data along the unlimited dimension using a hyperslab selection that corresponds to an HDF5 chunk.
3. A Reader opens the file and a dataset in a file, and queries the sizes of the dataset; if the extent of the dataset has changed, reads the appended data back.

Discussion points:

1. Since the new data is written to the file, and metadata update operation of adding pointer to the newly written chunk is atomic and happens after the chunk is on the disk, only two things may happen to the Reader:
    - The Reader will not see new data.
    - The Reader will see all new data written by Writer.

1.8 Appending a hyperslab

Description:

Appending a hyperslab that spans several chunks of a dataset with unlimited dimensions within a pre-created file and reading the new data back.

Goal:

Read data appended by the Writer to a pre-existing dataset in a file. The dataset has one or more unlimited dimensions. The data is appended by a hyperslab that is contained in several chunks (for example, appending 2-dim planes along the slowest changing dimension in the 3-dim dataset and each plane is covered by 4 chunks).

Level:

User Level

Guarantees:

- Readers will see the modified dimension sizes after the Writer finishes HDF5 metadata updates and issues H5Fflush or H5Oflush calls.
- Readers will see newly appended data after the Writer finishes the flush operation.

Preconditions:

- Readers are not allowed to modify the file.
- All datasets that are modified by the Writer exist when the Writer opens the file.
- All datasets that are modified by the Writer exist when a Reader opens the file.

Main Success Scenario:

1. An application creates a file with required objects (groups, datasets, and attributes).
2. The Writer opens the file and datasets in the file and starts adding data using H5Dwrite call with a hyperslab selection that spans several chunks.
3. A Reader opens the file and a dataset in a file; if the size of the unlimited dimension has changed, reads the appended data back.

Discussion points:

1. Since the new data is written to the file spans several chunks, and the metadata update operation to add a pointer to a newly written chunk is atomic and occurs after the chunks are written to the file, three things may happen to the Reader:
    a. The Reader will not see new data.
    b. The Reader will see one or more (but possibly not all) new chunks.
    c. The Reader will see all new chunks.
2. To avoid b. above, SWMR has to implement a “SWMR atomic operation” mechanism to assure that each H5Dwrite call for a particular object becomes an atomic operation, i.e., a Reader will see none or all of the new data. Implementing this capability is possible, but would need several new API calls, described in the appendices.

1.9 Appending n-1 dimensional planes

Description:

Appending n-1 dimensional planes or regions to a chunked dataset where the data does not fill the chunk.

Goal:

Read data appended by the Writer to a pre-existing dataset in a file. The dataset has one unlimited dimension. The data is appended by a hyperslab that leaves one or more chunks unfilled.

Level:

User Level

Guarantees:

- Readers will see the modified dimension sizes after the Writer finishes HDF5 metadata updates and issues H5Fflush or H5Oflush calls.
- Readers will see newly appended data after the Writer finishes the flush operation.

Preconditions:

- Readers are not allowed to modify the file.
- All datasets that are modified by the Writer exist when the Writer opens the file.
- All datasets that are modified by the Writer exist when a Reader opens the file.

Main Success Scenario:

1. An application creates a file with required objects (groups, datasets, and attributes).
2. The Writer opens the file and datasets in the file and starts adding data using H5Dwrite call with a hyperslab selection that does not fill the chunk.
3. A Reader opens the file and a dataset in a file; if the size of the unlimited dimension has changed, reads the appended data back.

Discussion points:

1. Since the new data is written to the file is smaller than a single chunk, it will not be propagated to the disk until the chunk is full or the Writer manually flushes the data.
2. In order to see the data as it is written, users will have to manually flush the data after each write. These files may grow to much larger sizes than equivalent non-SWMR files due to SWMR's current restrictions on recycling free space in the file.
3. The “SWMR atomic operation” mechanism discussed in the appendices is applicable for this use case too.

Appendices

In order to support SWMR access to HDF5 files in ways that are convenient to applications, we consider several changes to the HDF5 API covering improvements to the process of opening a file for SWMR reads and better support for appending to datasets.

Appendix A - SWMR File Access Model

The SWMR file access model follows the standard HDF5 model: the Writer and the Readers will need to indicate SWMR access using file access flags with the H5Fcreate and H5Fopen calls.

Writer Operations

The basic operations of a SWMR Writer are straightforward:

1. Call `H5Fcreate()` or `H5Fopen()` with the `H5F_ACC_RDWR` flag to create or open the file. This will make an annotation in the file that it has been opened for writing, but is not SWMR-safe.
2. Create HDF5 data objects and perform any non-SWMR-safe operations.
3. Begin SWMR writing by calling `H5Fclose()` followed by `H5Fopen()` with the `H5F_ACC_RDWR` and `H5F_ ACC_SWMR_WRITE` flags. This will make an annotation in the file that it has been opened for writing and is SWMR-safe.
4. Append or update data in the HDF5 file.
5. Call `H5Fclose()`.

Enhanced Writer Operations

In order to improve performance and usability of Writer applications, a new API routine[2] can be added to the HDF5 File API:

- `H5Fstart_swmr()` – A new file object routine that enables and disables SWMR access to an open file. Enabling SWMR access will write the `H5_OPEN_FOR_SWMR_WRITE` flag to the file and disabling it will remove the flag.

This routine could be used to set up a file with non-SWMR-safe operations, and then switch to SWMR-safe operations, as shown by the pseudo-code below:

```
/* Open the file, without the SWMR flag */
FILE_ID = H5Fopen("File.h5");

/* Perform non-SWMR-safe operations such as object creation */
…
```

[2] The name of the routine may change; it is used for illustrative purposes only.

```
/* Enable SWMR access */
H5Fstart_swmr(FILE_ID, TRUE);
/* Perform SWMR-safe operations */
…
/* Close the file */
H5Fclose(FILE_ID);
```

Having such a routine would allow a Writer to switch to SWMR access without closing & re-opening the file, retaining cached metadata and avoiding system call overhead.

Reader Operations

The basic operations of a SWMR Reader are even simpler:

1. Call `H5Fopen()` to open the file. This call will fail if the file has been opened for writing without the H5F_ACC_SWMR_WRITE flag.
2. Read data from the file.
3. Call `H5Fclose()`.

For the initial prototype, the Readers must open the file after the Writer has opened it for SWMR-safe writing. Files opened by for write access by HDF5 1.10 Writers will mark the file as either SWMR-safe or SWMR-unsafe (read access will not be marked). As a safety measure, H5Fopen calls made by Readers (either 1.8 or 1.10) will fail when the file is marked for SWMR-unsafe writes. Switching back to non-SWMR-safe writing will not be supported in the prototype.

Appendix B – Atomic File Object Operations

Situations can arise where a region in a dataset that spans several chunks corresponds to a logical "element" in the dataset. An example would be a three-dimensional dataset that stores 2D images with a third time dimension. If each plane is spread across more than one chunk, a reader may encounter partial data in SWMR if not all the chunks or chunk index nodes have been flushed to the disk.

To mitigate this, we propose adding a function to the HDF5 API that allows a user to more explicitly control the retention and flush behavior of specific HDF5 file objects:

- `H5Ocork()`[3] – A new generic object routine, which controls whether metadata for an object is held in the cache ("corked") or allowed to be flushed to the file.

This function could be used by an application to prevent cache flushes for a dataset until all the data has been written (and cached). At this point, `H5Oflush()` would be called to completely flush the metadata to the disk. Due to the way that HDF5 chunk index structures are updated under SWMR, this would ensure that readers would either see the entire slice or none of the slice, thus creating an "atomic operation" for the changes to the dataset[4]. The following pseudo-code shows its use:

```
/* Keep all metadata items that are relevant to a dataset in cache */
H5Ocork(DATASET_ID, TRUE);

/* Append a plane along the unlimited dimension */
/* Internally, this will consist of multiple data/metadata writes */
/* (Dataspace operations not shown) */
H5Dwrite(DATASET_ID,…, plane1);

/* Flush all modified metadata items for the dataset */
H5Oflush(DATASET_ID);
```

Although this code appears very similar to standard way of writing data to a dataset, the surrounding cork and flush calls ensure that the reader will not see part of a logical data element.

[3] There is already an `H5Orefresh()` routine targeted at readers, which refreshes an object's metadata.

[4] NOTE: Although this is similar in some ways to a transaction, it lacks some important semantics and we've avoided using the term in this document.

Appendix C – High-Level Dataset Append Function

Since SWMR writes will often consist of a long series of dataset append operations, we propose adding a convenience operation to the high-level HDF library that will condense the boilerplate dataspace operations and dataset write into a single function call:

- `H5DOappend()`[5] – A new "optimized dataset" routine, which both extends a dataset's dataspace in a particular dimension and writes data elements to the newly extended region in the dataset, eliminating much application code for performing similar actions.

The following pseudo-code shows its use:

```
/* Append a plane along the unlimited dimension */
H5DOappend(DATASET_ID,…, planeN);
```

Although this is very similar to the pseudo code in appendix B, the boilerplate dataset adjustments before the append would not be present in this case, saving developer time and allowing for optimization inside the library.

[5] Note that a similar routine for reading from the end of a dataset, "`H5DOpull`", would be a logical addition as well.

Appendix D – Append When More Than One Slice Fits in a Set of Chunks

In situations where several appended dataset slices or regions fit into a set of chunks, the writer will have to manually keep track of chunk boundaries in order to manually control dataset flush operations as outlined in Appendix B.

Although this could be done with the above functionality, the scheme still has some drawbacks for an application that would like to have chunks become visible without explicit knowledge of the chunk boundaries. An application that managed this knowledge would look something like this pseudo-code:

```
/* Determine chunk size */
H5Pget_chunk(DATASET_CREATION_PROPERTY_LIST_ID, &chunk_dims);

/* Get dataset dimensions */
SPACE_ID = H5Dget_space(DATASET_ID);
H5Sget_simple_extent_dims(SPACE_ID, …, &dims);

/* Keep all metadata items that are relevant to a dataset in cache */
H5Ocork(DATASET_ID, TRUE);

/* Append planes along the unlimited dimension */
for(n = 0; n < MAX; n++)
{
      H5DOappend(DATASET_ID,…, planeN);

      /* Increment the local copy of the dataset dimensions */
      dims++;

      /* If we are at chunk boundary, flush the dataset */
      if( dims % chunk_dims == 0)
      {
            /* Flush dataset */
            H5Oflush(DATASET_ID);

            /* Notify readers that dataset is updated */
            NotifyReader(DATASET_ID);
      }
```

```
}

/* Handle any partial chunks */

/* Flush all modified metadata items for the dataset */
H5Oflush(DATASET_ID);

/* Notify readers that dataset is updated */
NotifyReader(DATASET_ID);
```

To avoid having a user manage chunk boundary flushes (`H5Oflush` call above), we can add two properties that control the flush behavior, including optional callback functions that are invoked on appends and flushes:

- `H5Pset_append_flush()` – A new property that triggers two actions when a dataset append operation reaches a chunk boundary: calling an application callback routine when the chunk boundary is reached[6] and flushing the dataset to the file.
- `H5Pset_object_flush_cb()` – A new property which triggers an application callback when all the metadata for an object has been flushed to the file and the object is in a consistent state.

This addition to the HDF5 API allows for much simpler code, as shown in the pseudo-code below:

```
/* Set property to trigger callback when objects are flushed */
H5Pset_object_flush_cb(FILE_ID, &NotifyReaderCallback);

/* Keep all metadata items that are relevant to a dataset in cache */
H5Ocork(DATASET_ID, TRUE);

/* Set property to trigger callback and dataset flush at chunk boundaries */
H5Pset_append_flush(DATASET_ID, &UpdateAttributeCallback);

/* Append planes along the unlimited dimension */
for (n = 0; n < MAX; n++)
{
```

[6] This callback could be used to update an attribute attached to the dataset, for example.

```
        /* Note that any append which fills one or more chunks will trigger
           the library to call the UpdateAttribute callback and then flush the
           dataset, triggering the object flush callback
           (NotifyReader, in this case) after the dataset was flushed */
        H5DOappend(DATASET_ID,…, planeN);
    }

    /* Handle any partial chunks */

    /* Modify an attribute, for any partial chunks */
    H5Awrite(ATTR_ID,…, &attr_value);

    /* Flush all modified metadata items for the dataset */
    /* (Also triggers object flush callback) */
    H5Oflush(DATASET_ID);
```

These changes would allow an application to set up actions that occur when various events occur (appending across a chunk boundary, flushing an object) within the HDF5 library, while keeping an application's monitoring of internal HDF5 state to a minimum.

Revision History

February 14, 2013	Version 1 circulated for comment within The HDF Group.
February 18, 2013	Version 2 addressed the HDF developers' comments and sent for final review.
February 19, 2013	Version 3 sent to Nick Rees, DLS.
February 27, 2013	Version 4 fixed minor types, added a case of pre-existing attribute and multiple unlimited dimensions; sent to Nick Rees, DLS, and the HDF developers.
March 3, 2013	Version 5 added use case 3.5 and fleshed out the programming model a bit. Circulated for comment within The HDF Group.
March 5, 2013	Version 6 accepted Quincey and Dana's comments and added transaction mechanism example.
March 7, 2013	Version 7 added section 5; sent to Nick Rees, DLS
March 12, 2013	Version 8 converted sections 4 and 5 to appendices and changed the transaction nomenclature. Sent to THG for comments.
March 13, 2013	Version 8 reformatted; sent to Nick Rees, DLS.