Design: File Locking under SWMR - Semantics, Programming Model, and Implementation

Vailin Choi
Dana Robinson

Under the new single-writer/multiple-reader (SWMR) feature of version 1.10.0 of the HDF5 Library, file locking will be used to help prevent inappropriate concurrent file access. This locking is only intended to help the user conform to SWMR semantics and is not a fundamental part of SWMR synchronization.

This document describes the semantics of the file locking operations and programming model. The various system-level file locking calls that will be used are discussed as well as some implementation details. The h5clear tool is also described.

1 Introduction

The 1.8.x version of the HDF5 Library does not detect concurrent file access by multiple writers and readers. Since the HDF5 Library is a straightforward input/output (I/O) library and not a server or other central data store, no state is shared between reader and writer processes. Nor does the HDF5 Library support any sort of inter-process communication (IPC) between coordinating processes. This can cause problems when multiple processes open a file and at least one will modify data. Obviously, there would there be problems due to multiple non-coordinating writers accessing a file, but difficulties can arise even when only one writer is modifying the file. This is due to the internal caching layers of the library and the structure of HDF5 files, which contain a number of file metadata objects which contain offsets that are used to locate data and other file metadata. When some of these metadata objects have not yet been flushed to disk, the on-disk file will contain an incomplete picture of the file, possibly including some offsets that point to unflushed disk regions containing garbage. When these garbage regions are read from a disk by an independent reader process, the library will usually fail.

The new SWMR feature in HDF5 1.10.0 mitigates the concurrent access problem in a number of ways. The most important changes were to the cache layers, which ensure that the on-disk structures are always complete and valid, if not perfectly up-to-date, as well as adding SWMR status flags to the HDF5 file's superblock. However, correct library behavior still requires the user to be careful about concurrent file access. For example, if a reader process opens a file before a writer process and the writer performs operations that are not supported under the current SWMR implementation (for example, dataset or group creation), the reader process could encounter invalid data and crash.

In order to help users achieve correct SWMR behavior ("SWMR semantics"), we will implement a file locking scheme that will attempt to prevent problematic file accesses. Although not foolproof, particularly with respect to older versions of the HDF5 Library, the file locks will help ensure correct use of this somewhat complicated feature.

Note that this file locking scheme should not be taken to imply that the HDF5 Library uses region locks to implement SWMR. Aside from the initial file open, locks are not used to manage file I/O between the SWMR processes.

2 File Locking Semantics under SWMR

The particular semantics of file locking will depend on and be limited by a system's available file locking API calls. The particular system calls will be determined at compile time; however, we will endeavor to use file locks that roughly correspond to BSD-style locks based on flock(2). See the implementation notes and appendices below for more information.

2.1 File Exclusion Mechanisms

The HDF5 Library will employ two means to regulate access to HDF5 files under SWMR.

1. File locking API calls to apply or remove an advisory lock on an open file. Note that these will also be used for non-SWMR access as a way to prevent inappropriate file access such as might occur with two writer processes.

    a. Files will be locked during the H5Fopen() or H5Fcreate() call.

    b. Locks can be shared (read) or exclusive (write).

    c. Locks will lock the entire file, not regions in the file.

    d. When non-blocking lock calls are available, locks will not block.

    e. Locks will be released automatically when the file closes. Alternatively, the user can unlock the file using the system's unlock call; however, care will have to be taken to match the unlock call with the HDF5 Library's file locking scheme.

2. Setting a flag in the file's superblock to mark the file as open for writing.

    a. The library will mark the file when opened for writing based on file open access flags. This will happen for both SWMR and non-SWMR reading. This marking ensures file consistency for concurrent accesses.

    b. The library will clear the flag when the file closes.

2.2 Writer Actions

Without SWMR. When a writer process creates or opens a file **without** SWMR, it will:

- Place an exclusive lock on the file—the file will remain locked until it closes
- Ensure the file's superblock is not already marked for writing or SWMR writing mode
- Mark the file's superblock for writing mode

SWMR. When a writer process creates or opens a file **with** SWMR write access, it will :

- Place an exclusive lock on the file
- Ensure the file's superblock is not already marked for writing or SWMR writing mode
- Mark the file for writing and SWMR writing mode
- Release the lock before returning from *H5Fopen or H5Fcreate*

Once a writer successfully opens or creates a file, no other writer can open or create the file (with or without SWMR). Readers cannot access the file either with one exception: if a writer has successfully opened the file with SWMR write, then a reader can access the file with SWMR read.

2.3 Reader Actions

Without SWMR. When a reader process opens a file **without** SWMR, it will:

- Place a shared lock on the file
- Ensure the file is not already marked for writing or SWMR writing mode

SWMR. When a reader process opens a file **with** SWMR, it will:

- Place a shared lock on the file
- Ensure the file is marked in writing and SWMR writing mode

Once a reader successfully opens a file, no writer can open the file (with or without SWMR), but other readers can open and access the file (with or without SWMR).

2.4 HDF5 Library Compatibility

HDF5 1.10.x	Full SWMR; respects locks and superblock.
HDF5 1.8.y (y = TBD)	Will respect locks, but will not participate in SWMR.
HDF5 1.8.0 - 1.8.(y-1)	Will ignore SWMR. All file open/create succeed.

2.5 SWMR Compatibility Matrices

The table below depicts the result of a second file open after the first open succeeds—file opens by writer and reader with combinations of versions and access flags.

2.5.1 1st Open 1.10 / 2nd Open 1.10

		First Open/Create (1.10)			
	file open access flags	*read*	*write*	*SWMR read*	*SWMR write*
Second Open (1.10)	*read*	succeed	fail	succeed	fail
	write	fail	fail	fail	fail
	SWMR read	succeed	fail	succeed	succeed
	SWMR write	fail	fail	fail	fail

2.5.2 1st Open 1.10 / 2nd Open 1.8.y+

		First Open/Create (1.10)			
	file open access flags	*read*	*write*	*SWMR read*	*SWMR write*
Second Open (1.8.y+)	*read*	succeed	fail	succeed	fail
	write	fail	fail	fail	fail
	SWMR read	N/A	N/A	N/A	N/A
	SWMR write	N/A	N/A	N/A	N/A

2.5.3 1st Open 1.8.y+ / 2nd Open 1.10

First Open/Create (1.8.y+)

Second Open (1.10)	*file open access flags*	*read*	*write*	*SWMR read*	*SWMR write*
	read	succeed	fail	N/A	N/A
	write	fail	fail	N/A	N/A
	SWMR read	succeed	fail	N/A	N/A
	SWMR write	fail	fail	N/A	N/A

2.5.4 1st Open 1.8.y+ / 2nd Open 1.8.y+

First Open/Create (1.8.y+)

Second Open (1.8.y+)	*file open access flags*	*read*	*write*	*SWMR read*	*SWMR write*
	read	succeed	fail	N/A	N/A
	write	fail	fail	N/A	N/A
	SWMR read	N/A	N/A	N/A	N/A
	SWMR write	N/A	N/A	N/A	N/A

3 Programming Model

General SWMR operations are discussed in “The Programming Model” section in the SWMR user's guide. [10]

3.1 Determining the Success of Open/Create

When attempting to open an HDF5 file, the possibility exists that the file will be locked by another process in a way that prevents access. If this occurs, the H5Fopen() or H5Fcreate() call will fail and a negative hid_t value will be returned to the caller. In order to determine if the failure was due to an inability to acquire the lock or for some other reason, the H5E (error) interface can be used to query the error code set by the last API call[1].

To query the error code, you will need to create a callback function which will be passed to H5Ewalk() along with the error stack obtained from H5E_get_current_stack(). This simple callback function will only need to inspect the major and minor error codes in the passed-in H5E_error2_t structure and return its findings via the client_data pointer (for example, via a Boolean indicating that a file lock failure was found someplace in the error stack). The major code will be H5E_FILE and the minor code will be H5E_CANTLOCK.

See section 9 of the HDF5 User's Guide for more details on HDF5 error handling.

Note that our implementations of file locking will use non-blocking system functions when available.

[1] Note that, unlike errno, we use thread-specific error stacks so this functionality works in thread-safe builds of the library.

4 Platform-specific Implementations

The usual platform-independent layer in the HDF5 Library will be used to hide the underlying implementation of file locking on a particular platform. In all cases, locks will be per-file and advisory; the library will respect them, making them "mandatory" as far as a SWMR-aware HDF5 Library is concerned. All locking implementations will endeavor to provide BSD lock capabilities and semantics (see below).

In a SWMR-aware library (1.10.*x*, 1.8.*y* (y has not yet been determined)), file I/O on active SWMR files will require file locking on a supported platform. When no suitable system file locking API calls are available, the HDF5 Library will use a dummy file locking function that always fails, preventing concurrent SWMR file access[2].

4.1 POSIX Systems

When available, BSD-style locks based on flock(2) will be used for file locking. When this function is not present, POSIX-style locks based on fcntl(2) will be used instead. The particular function used will be determined at configure/build time using the detection features of Autotools or CMake. lockf(3) is another possible lock function found on POSIX systems; however, lockf(3) usually uses the same underlying mechanisms as fcntl(2) locking, and we have not identified any POSIX system on which fcntl(2) is not present.

The primary reason for this preference for BSD locks is that POSIX locks will be lost if *any* file descriptor for the file is closed, which could cause problems in large applications where another part of the program might access the file. BSD locks do a much better job of providing per-file lock semantics, and we do not need the finer-grained locking functionality of the POSIX locks.

See the “Appendix” chapter beginning on page 14 for more information on BSD and POSIX locking.

The availability of the various POSIX and BSD lock functions on currently supported operating systems is as follows (recent OS versions, as of 2015):

OS	flock	fcntl	lockf
Linux	Y	Y	Y
FreeBSD	Y	Y	Y
OS X	Y	Y	Y
Solaris	N	Y	Y
Cygwin	Y	Y	Y

4.2 Windows

NOTE: Even though we currently do not support SWMR on Windows, the win32 implementation of file locking is included for completeness.

[2] We expect this to be very rare. POSIX locks via fcntl(2) are almost always available on any platform that is likely to perform SWMR I/O. If a use case demanded it (perhaps on embedded hardware), this could easily be changed. For the time being, however, enforcing SWMR file I/O ordering via locks seems likely to increase the odds of successful use of the feature.

Windows uses the LockFile() and LockFileEx() functions for file locking (the Ex function has more advanced features). The Ex call has semantics that are very similar to those of BSD locks but also has functionality that is available via POSIX locks, such as region locking, should that ever be required. As in the POSIX and BSD locks, both shared and exclusive locks are available, and locks can be either blocking or non-blocking.

5 The h5clear Tool

As stated in the file locking semantics section, the library will add a flag to the file's superblock to mark that a file is in writing or SWMR writing mode, and it will clear the flag when the file closes. However, a situation such as a program crash or power outage may occur where an open file is closed without going through normal library file closing procedure, and the superblock flag will not be cleared as a result. This situation will prevent a user from opening the file with a SWMR-aware HDF5 Library. The library will therefore provide a simple tool called *h5clear* so that a user can clear this flag.

The tool will open the input file with a file access property list that is set with a private property called H5F_ACS_CLEAR_STATUS_FLAGS_NAME. Upon encountering this property during file open, the library will clear this flag in the file's superblock. The tool will then close the file.

6 Acknowledgements

This work is supported by a customer of The HDF Group, Dectris.

7 Revision History

January 10, 2014	Version 1 circulated for comment within The HDF Group SWMR team.
September 2, 2015	Version 2 is a complete rewrite of Vailin's original document. Circulated for comment within The HDF Group SWMR team.
September 21, 2015	Version 3 filled in the tables and added the putative H5LTcheck_lock_error() API call RM page. Circulated for comment within The HDF Group SWMR team.
September 28, 2015	Version 4 rearranged the tables and added a few other minor edits. Circulated for comment within The HDF Group SWMR team.
March 31, 2016	Version 5. Editing to prepare document for posting on the website.

8 References

1. The HDF Group. "Single-Writer/Multiple-Reader (SWMR) Documentation" https://www.hdfgroup.org/HDF5/docNewFeatures/NewFeaturesSwmrDocs.html (September 1, 2015).
2. The HDF Group. "HDF5 User's Guide: Error Handling" https://www.hdfgroup.org/HDF5/doc/UG/HDF5_Users_Guide-Responsive%20HTML5/index.html#t=HDF5_Users_Guide%2FErrorHandling%2FHDF5_Error_Handling.htm (November 2015).
3. Pid Eins. "On the Brokenness of File Locking" http://0pointer.de/blog/projects/locking.html (June 26, 2010).
4. Pid Eins. "Addendum on the Brokenness of File Locking" http://0pointer.de/blog/projects/locking2 (June 28, 2010).
5. MSDN. "LockFile function" https://msdn.microsoft.com/en-us/library/windows/desktop/aa365202(v=vs.85).aspx (accessed August 4, 2015).
6. MSDN. "LockFileEx function" https://msdn.microsoft.com/en-us/library/windows/desktop/aa365203(v=vs.85).aspx (accessed August 4, 2015).
7. man7.org Linux man pages. "flock(2)" http://man7.org/linux/man-pages/man2/flock.2.html (September 21, 2014).
8. man7.org Linux man pages. "lockf(3)" http://man7.org/linux/man-pages/man3/lockf.3.html (March 2, 2015).
9. man7.org Linux man pages. "fcntl(2)" http://man7.org/linux/man-pages/man2/fcntl.2.html (May 7, 2015).
10. The HDF Group. “HDF5 Single-Writer/Multiple Reader User’s Guide.” https://www.hdfgroup.org/HDF5/docNewFeatures/SWMR/HDF5_SWMR_UsersGuide.pdf (March 2016).

9 Appendix

9.1 flock(2) Semantics ("BSD Locks")

In brief:

- Specified in 4.4BSD, though widely available. Occasionally implemented using the same primitives as fcntl(2), *which implies those semantics - see below!* (NOTE: Linux does not do this).
- Allows getting, setting, and testing for the existence of both shared (read) and exclusive (write) locks.
- Locks are advisory.
- Locks are per-file. They cannot cover a particular region in the file.
- Any number of processes can hold a shared lock. Only one process can hold an exclusive lock. A single process cannot hold both a shared and exclusive lock for a file.
- Locks can be set to block or return immediately on failure to acquire.
- Locks are bound to a process and an open file table entry. File descriptors are considered independent and handled independently (in other words, a process with two file descriptors to the same file can block its own locks). File descriptors created via dup() or fork() are considered identical and will be closed when all duplicated descriptors are closed or an unlock operation takes place on any descriptor.
- Any lock can be placed on a file regardless of how it was opened (read/write).
- Locks are preserved across fork() and execve().
- NOTE: BSD and POSIX locks do not necessarily interact. In other words, a BSD lock is not guaranteed to be respected by a call to fcntl(2).

9.2 fcntl(2) Semantics ("POSIX Locks")

In brief:

- Specified in SVr4, 4.3 BSD, and POSIX.1-2001 (very widely available).
- Allows getting, setting, and testing for the existence of both shared (read) and exclusive (write) locks.
- Locks are advisory.
- Locks can cover regions of the file at byte-level granularity. The entire file can also be locked.
- Any number of processes can hold a shared lock. Only one process can hold an exclusive lock. Only one type of lock can be held for a region, though a process may hold both shared and exclusive region locks throughout a file.
- Locks can be set to block or return immediately on failure to acquire.
- Exclusive locks require a file being opened in write or read-write mode, shared locks require a file being opened in read or read-write mode.

- Locks are bound to a process and an inode (*not* a file descriptor). This means that a process closing any file descriptor for the file will release all fcntl locks. This can be a problem if another part of the code opens and closes a file for any reason, say, as an existence test.
- Locks are not preserved across fork() but are preserved across execve().
- POSIX locks have trouble with stdio buffering and will be disabled in the stdio VFD.
- Mandatory locks are non-POSIX. The Linux implementation is known to have severe race conditions. All mandatory locking has corner cases that allow bypassing the lock (for example, unlink() often ignores mandatory locks). File systems must often be specially mounted to take advantage of them.
- Note: our implementation ignores fcntl64(2) on Linux, since this is handled transparently.
- lockf(3) is usually implemented using the same primitives as fcntl(2) locking.
- NOTE: BSD and POSIX locks do not necessarily interact. In other words, a BSD lock is not guaranteed to be respected by a call to fcntl(2).

9.3 H5LTcheck_lock_error

Name: H5LTcheck_lock_error

Signature:

```
herr_t H5LTcheck_lock_error(hbool_t clear_stack,
                            /*OUT*/ hbool_t *had_lock_error)
```

Purpose:

Determine if a file lock access error has occurred.

Description:

The HDF5 Library uses file locks to help users get the ordering of single-writer/multiple-reader (SWMR) writers and readers correct. If a SWMR open fails due to a lock access problem, this API call can be used to see if the access produced an error.

Notes:

This function does not automatically clear the HDF5 error stack. If this API call is to be called after a previous file lock error, the stack must be cleared either by setting clear_stack to TRUE or by using H5E_get_current_stack and passing the returned hid_t to H5Eclear. If this is not done, the API call will detect the previous error in the stack.

Parameters:

hbool_t	clear_stack	IN: Whether the error stack for this thread should be cleared.
hbool_t	had_lock_error	OUT: If a lock error has occurred.

Returns:

Returns a non-negative value if successful. Otherwise returns a negative value.