Making H5FD Multi-Thread Safe:
A Sketch Design

John Mainzer
Lifeboat, LLC

11/30/22

Introduction

H5FD is fundamentally a straight forward module – thus the majority of its multi-thread issues are inherited either from its dependencies or the underlying storage system.

As this is a preliminary sketch design for retrofitting multi-thread safety on H5FD, the initial objective is to outline the issues to be addressed, and to outline possible solutions. A more thorough review of the code will be necessary before this sketch design can be brought to something approaching final form.

A Quick Overview of H5FD

The HDF5 library maintains an abstraction layer at the bottom of the library known as the VFD (Virtual File Driver) layer. Its purpose is to present the underlying file(s) as an extensible vector of bytes to the higher levels of the library. Historically, this abstraction layer has been used to allow the HDF5 library to run on different operating systems with different file I/O APIs, to simulate large files on file systems with a 2 GB max file size, and to segregate metadata and raw data into separate files. More recently it has been used to implement sub-filing, mirror HDF5 files on remote systems as they are being written, version HDF5 files, and implement a new, more general version of SWMR.

In discussing this layer in the library, it is important to distinguish between the pass through and terminal Virtual File Drivers (VFDs) and the code that supports them – the VFD layer proper, which is the topic of this document.

Terminal VFDs handle I/O to the specified file or storage device. Pass through VFDs operate on I/O requests but do not do I/O directly. For example, the sub-filing VFD routes I/O requests to the appropriate I/O concentrator(s) and hence to the target sub-file(s), and the splitter VFD duplicates all write requests and routes them to the mirror VFD so it can transmit them across the network and apply them on the target remote system.

In contrast, the VFD layer handles the following matters:

- On file open or creation, it instantiates the stack of VFDs specified in the FAPL (File Access Property List), and routes calls from one VFD to the next VFD further down in the stack.

- Once the file is open, it routes I/O and control calls to the top level VFD, which then percolate down through the VFD stack as required.

- After file close, it manages shutdown and discard of the VFD stack.

- With the addition of support for pluggable VFDs, it manages the load of plug-able VFDs from file system via the H5PL (pluggable) module.

- Finally, it makes all of these capabilities accessible via the H5FD public API

As shall be discussed, the VFD layer proper makes heavy use of a variety of HDF5 packages – and thus most of its multi-thread safety issues are addressed by work on those packages.

Obviously, making the VFD layer proper multi-thread safe is useless without a thread safe VFD. How difficult this is depends on what the VFD does, and in the case of terminal VFDs, on the underlying storage system. This will vary, but as the initial target of interest will be POSIX and the sec2 VFD, note that sec2 VFD already uses the pread() / pwrite() and thus should require only minor modification to become fully thread safe once H5E, H5I, and H5P are thread safe.

Multi-Thread Issues in H5FD

The multi-thread issues in H5FD are of two types: use of other, potentially non-multi-thread safe packages, and issues internal to H5FD proper. Before discussing how these challenges might be addressed, it will be useful to discuss these issues in greater detail.

Use of other HDF5 packages by H5FD

H5FD makes use of the following packages:

- H5MM
- H5FL
- H5E
- H5I
- H5P
- H5CX
- H5F
- H5PL

As usual, issues with H5MM and H5FL may be avoided by using C memory allocation functions directly, and by either not maintaining free lists, or doing so internally. With regards to free lists, the point appears to be moot, as while a free list for instances of H5FD_free_t is declared, the structure does not appear to ever be allocated.

Once they are made multi-thread safe, calls to H5E, H5I, and H5P are non-issues, as long as H5FD can refrain from holding locks during these calls. There are calls to H5I_iterate() in H5FDint.c, which should be replaced with more thread friendly calls once these are added to H5I. Note that these calls are associated with driver registration – of which more later.

H5CX has not been investigated yet. However, the H5CX calls used by H5FD are all involved in getting and setting the DXPL ID in the thread specific context. Thus, for purposes of H5FD thread safety, these calls are not an issue.

Making H5F multi-thread safe is not on the agenda any time soon, and thus use of H5F is a potential problem. Fortunately, all but one of the uses of H5F are either references to constants, or invocations of simple macros which are easily ruled thread safe by inspection. (i.e. H5F_addr_defined(), H5F_addr_eq(), etc.) The one exception is the call to H5F_eoa_dirty(), which is called whenever file space is allocated or de-allocated at the VFD layer. These calls dirties the super block, and either the driver information block or the super block extension depending on the version of the superblock, and the VFD used. Making this call thread safe is well beyond the scope of the current effort, and thus the call will have to be modified to obtain the global lock if it is not held already.

Finally, there is the matter of H5PL – the package that handles pluggable filters, VOL connectors, and VFDs. Not listing this package in our initial list of essential multi-thread conversions may have been an oversight. That said, it is called so infrequently, that we may be able to get away with leaving it below the global lock for now. Certainly, in the case of H5FD, we can avoid the issue temporarily by using only built in VFDs.

Multi-thread thread issues in H5FD proper

A review of the H5FD public, private, and developer APIs reveals the following multi-thread safety issues in the current H5FD implementation.

The H5FD_file_serial_no_g Global Variable:

H5FD assigns a serial number to each VFD instance it creates. To avoid duplicate serial numbers, some sort of mutual exclusion is required for this operation. Making H5FD_file_serial_no_g into an atomic variable would probably be sufficient.

H5FD_t and its Derivatives:

When an instance of a VFD is created, an associated structure is allocated, whose first field is an instance of H5FD_t. There are a number of fields associated with this structure that may be problematic from a multi-thread perspective.

First, most if not all terminal VFDs will have an EOF (End of File) and an EOA (End of allocation) associated with them. The values of these fields will be changed via VFD API calls such as get/set_eof() and get/set_eoa(). These calls may be made directly, or more typically, through file space allocation / free calls. Since there are no prohibitions on multiple such calls being active simultaneously, some form of mutual exclusion is required to protect these values and prevent corruption.

Second, the multi-thread paradigm allows multiple threads to use these structures simultaneously. With the above exceptions, the fields of these structures are typically constant after file open – which suggests that making a few fields into atomic variables will solve the obvious problem. However, there is also the matter of not freeing these structures out from under other threads on file close. An active thread counter combined with a closing flag will probably be sufficient here.

VFD Registration and Deregistration:

With the addition of support for plug-able VFDs, the VFD layer proper has acquired more complex code supporting VFD registration and deregistration. This code makes heavy use of H5PL (the package supporting pluggable filters, VOL connectors, and VFDs), so it is probably best to side step this issue until there is at least an initial review of that package, and the issues involved in making that package multi-thread safe.