Analysis of Boost Memory-Mapped Files

ABSTRACT

Boost's Memory-Mapped Files implementation is a compent, thin veneer over the underlying OS functions. It provides the intersection of functionality between Windows and POSIX, with no access to the underlying file descriptors. Because it prevents full exploitation and control of map that the OS provides, it isn't well suited to use for the write-ahead log.

Features

The `mapped_file` class offers these methods:

`constructor`
`open`
`is_open`
`close`
`size`
`data`
`const_data`
`alignment`

Only functionality available on both operating systems is supported. Note the lack of a `handle` method or similar. (This is consistent with `iostreams` but in this case prevents necessary access.)

One benefit of using Boost is consolidated error handling. I assume it's well done, but it seems to me that this critical function needs careful and specific treatment.

Questionable choices

In both Windows and Linux, the map function — `mmap` or `CreateFileMappingA` & `MapViewOfFileEx` — are passed hard-coded flags based on the value of the `readonly` member. If the mapping function fails for any reason, the functions are calls again without the flags. That this done is not documented; why it is done doesn't merit a comment in the source code. In consequence, the attributes of the mapping are nondeterministic and, moreover, cannot be ascertained from the object.

Unsupported Features

msync POSIX `msync` permits committing an arbitrary range of pages to disk. Windows has a similar function, `FlushViewOfFile`. This feature and absolutely necessary to support user-requested *flush* functionality.

madvise POSIX `madvise` lets the application advise the kernel of the intended use of the mapped file. Chief among these for our purpose is `MADV_SEQUENTIAL`, because the file will be accessed for the most part sequentially. Windows offers no similar function, but `madvise` might well turn out to be important for performance on Linux.

shared memory
Both POSIX and Windows provide for a memory-mapped file to be shared between processes. This could prove to be useful in limited ways. For instance, the `wald` daemon that creates new WAL files could map the file into memory before advising the `Wal` object that the file is ready,

saving the Wal object the time to map the file. (It would only gain access to already-mapped pages.) It also seems possible that the Worker would write to the WAL file and that only synchronization would fall to the `Wal` object.

Alternative design

My recommendation is to develop a POSIX-like C library of missing functions for Windows. I would make it open source to allow others to offer advice and fixes. For mapped files, there are only a few functions needed and implementation simple. The flags are differently named, but are semantically equivalent (and, in any case, any differences are also to be found in Boost's implementation).

By creating a C library, the same WAL implementation would compile in Linux and Windows. Problems would be directly addressable, and the reference implementation available for comparison.

Having created a compatible C library, should a C++ object such as Boost's be created as a wrapper? If so, Boost's own implementation could be used, less the Windows code and with the above missing functionality added. Because RAII cannot be applied to this object in this application, I wouldn't bother to create the wrapper unless we decided to implement I/O operators `operator<<` and `operator>>` for I/O with the Thrift classes, in which case the mapped-memory object could act as source and sink. If the Thift classes will serialize to/from a `char*` pointer, it's hard to see any reason to write a wrapper class.

Below is a list of the needed functions.

POSIX	Windows
open	CreateFile†
lseek	LZSeek†
close	FileClose†
mmap	CreateFileMapping, MapViewOfFileEx
madvise	(not possible)
msync	FlushViewOfFile
munmap	UnmapViewOfFile, CloseHandle∗

Because Windows already has some of these functions, wrappers — perhaps called mopen, mlseek, mclose — would be needed.

The mapped-memory functionality in Windows is, like a lot of Windows' functionality, similar to POSIX and different enough to be a nuisance. Implementing a POSIX version is perhaps one day's work. It would provide a more powerful and flexible interface to the operating system, and remove the uncertainty and limitations inherent in the Boost implementation.

† The POSIX function exists in Windows, but the `FILE*` handle cannot be used with memory-mapped files.

∗ CloseHandle must be called twice, once for the file handle, and once for the handle to the mapping object.