Are views of memory-mapped files coherent within a single process? (And how this was the wrong question.)

devblogs.microsoft.com/oldnewthing/20151218-00

December 18, 2015

Raymond Chen

A customer wanted some clarification. An MSDN article says

> This means that changes made to a page in the memory-mapped file via one process's view are automatically reflected in a common view of the memory-mapped file in another process.

"The above paragraph says that the views are coherent across processes, but what about within a single process?"

Yes, the views are coherent within a single process, too. The documentation for `MapViewOfFile` says

> … file views derived from any file mapping object that is backed by the same [local] file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The customer explained that they had a large file that they wanted to update from multiple threads. No two threads operate on the same bytes of the file, but the pieces are very close together and sometimes are interleaved. (As an extreme example: Thread 1 is operating on all the bytes at odd file offsets, and thread 2 is operating on all the bytes at even file offset.) Right now, they are using critical sections to make sure that only one thread updates the file at a time. They were thinking of using a memory-mapped file and letting each of the threads party on the portion of the view that corresponds to the work that that thread is doing. Since the threads are operating on disjoint portions of the file, they wouldn't need to synchronize access with each other. They just want to make sure there are no hidden gotchas with coherency.

If you look at their problem description, you'll see that view coherency is totally not the issue at hand. They aren't creating multiple views. They are using a *single* view and sharing it among multiple threads. This is not about view coherency: There is only one view, so it is trivially coherent with itself. This is really about memory coherency. So let's take the view out

of the picture. The actual question is "I have a chunk of memory that multiple threads are updating without locking. No two threads are updating the same byte of the memory. How safe is that?"

The answer is "Sort of, but you need to be careful."

The solution is to use atomic memory operations. But we're not really interested in the atomicity, seeing as each byte is operated on by only one thread. What we really care about is that one thread's writes don't incidentally write to bytes that belong to another thread.

In practice, this means operating on objects that can be updated atomically by the processor: Win32 guarantees atomic access for properly-aligned 32-bit values and properly-aligned pointer-sized values.

Therefore, you can slice up the file into chunks so that everything you operate on are either properly-aligned 32-bit values or properly-aligned pointer-sized values (though why you are putting pointers in a file eludes me), then you can have all the threads access the memory directly, and they won't step on each other. It's okay if a single slice consists of, say, four 32-bit values that are individually aligned, although the slice as a whole is not 32-byte aligned. The point is that when you access the slice, you access in pieces that are atomically updatable.

But if you are slicing up the file into bytes, like in our example above, then you cannot just have them all modifying bytes freely because byte-granular access is not guaranteed to be atomic. Windows can run on systems that do not support byte-granular access, such as the original Alpha AXP. Writing a byte on the original Alpha AXP was a multi-step affair: Load the eight-byte-aligned chunk of memory surrounding the byte into a single 64-bit register, update the single byte in that register, then write all eight bytes back to memory. Observe that this update is not atomic: If two threads try to write different bytes that happen to reside in the same eight-byte-aligned chunk, the writes may collide, and one of the writes will appear to be be lost. (The Alpha AXP also has a notoriously weak memory model.)

Bonus chatter: If you wanted to avoid having to rewrite the program too much, you could use the `WriteFile` function with an `OVERLAPPED` structure that provides the explicit offset for each write. This avoids having to synchronize the threads so that two threads don't try to move the file pointer at the same time. On the other hand, you aren't avoiding the synchronization entirely because I/O to a synchronous file handle is serialized. To get truly parallel writes, you need to open the file in `FILE_FLAG_OVERLAPPED` mode, but that is potentially a bigger change to the program.

Raymond Chen

Follow