Trying to allocate the same virtual address in multiple processes

devblogs.microsoft.com/oldnewthing/20181121-00

November 21, 2018

Raymond Chen

A customer wanted to know if it is possible to allocate shared memory at the same virtual address in multiple processes.

The short answer is that you can try to map a shared memory block at a specific address by using the `MapViewOfFileEx` function, but there is no guarantee that the desired address will be available.

You could use the `VirtualQueryEx` function to explore the address spaces of each of the participating processes, and then use the `MapViewOfFile2` function to map the view into each of them. If any of the mapping calls fails (say, due to a race condition), then roll back and look for another address range. Repeat until successful, or you run out of addresses to try.

If processes can join the party dynamically, you can run into the problem where the existing processes agreed on an address to use, and began using that memory, and then a new process joins in, but the address is not available in that new process.

We asked the customer why they wanted to do this.

The customer explained that they had a main program that wanted to expose some data to diagnostic tools. If they could put the shared memory block at the same address in every process, then they could put some `std::vector` and `std::map` objects in the shared memory. Then they would have all the conveniences that come with having ready-made `std::vector` and `std::map` implementations.

Okay, so now that we understand the scenario, we see that even if they managed to get the same virtual address in every process, it still wouldn't work.

For one thing, the default allocator for `std::vector` and `std::map` obtains memory from the C++ runtime, which will probably allocate it from the heap. That heap is not part of the shared memory, which means that the objects contained by the vector or map are not in the shared memory block, so they aren't being shared at all in the first place.

You could try to fix this by creating a custom allocator that allocates memory out of the shared memory block, but now you've signed up for writing a custom allocator.

The second problem is thread safety. The diagnostic process needs to make sure not to try to read the vector or map while the main process is mutating it. The data races could result in accessing memory that has been freed. You could use a mutex to protect access to the vector and map, but you would either have to expand the scope of the mutex to cover all accesses, or build a cross-process reader/writer lock so that the diagnostic processes can take a read lock, while the main process uses a write lock when mutating.

The third problem is that the main process and the diagnostic processes will have to use the same versions of the C++ runtime library. If they fall out of sync, then you have wandered into undefined behavior.

My impression is that the customer was considering opening the diagnostic interfaces to third parties so they could write fancy diagnostic tools. In that case, can easily have the main process and diagnostic processes falling out of sync.

Creating a cross-process data structure is quite complicated. You can't just put a `std::vector` in some shared memory and expect everything to be all peachy.

Raymond Chen

Follow