How can I reserve a range of address space and create nonzero memory on demand when the program reads or writes a page in the range, even when multithreading?

devblogs.microsoft.com/oldnewthing/20180126-00

January 26, 2018

Raymond Chen

Last time, we described how you can become the page access manager for a range of pages, but it required that all the accesses came from one thread at a time because you don't want another thread to be able to access the memory while it is still being prepared. That requirement exists because we are preparing the pages in place, and once you unprotect the page so you can prepare the page, another thread can sneak in and see the pages before they're ready.

Let's see what we can do to get this to work in the multithreading case, too.

Unfortunately, I don't see a version of `VirtualAlloc` that lets you say, "Please take this page of memory I already have and map it into that location over there." You can do it if you're willing to use AWE, but that requires permission to allocate physical memory, and you lose the ability to write-protect pages (which makes detecting dirty pages harder), and it works only on natively 32-bit versions of Windows.

So we'll have to use a different trick: mapping the same block of memory into two locations. We'll take the trick a step further and map the same memory twice, but with different permissions.

First, create a shared memory block with `CreateFileMapping`, specifying a page protection of `PAGE_ READWRITE`. This gives you read/write access to the underlying memory.

Next, map the shared memory block with `MapViewOfFile`, specifying a file mapping access of `FILE_ MAP_ WRITE`, since we will eventually want to give the client write access (just not at first). This is the memory region that will be used to hold client-visible memory. Right now, it's filled with zeroes, but we'll fix that soon.

Use `VirtualProtect` to change the page protection to `PAGE_ NOACCESS` for all the pages. This removes access to all the pages. The client-visible memory is now ready.

When an access violation occurs and you want to swizzle some memory and map it in, here's what you do:

Use the faulting address to figure out which page of data needs to be swizzled and mapped in.

Use some sort of synchronization to make sure only one thread is doing the swizzling for this page. If you discover that the page has already been swizzled, then you are done because the other thread already did the work for you.

Otherwise, you are the first thread to handle the access violation. Find the corresponding page in your file mapping and use `MapViewOfFile` with a file mapping access of `FILE_MAP_WRITE`. This creates a second view of the page in which the client just took an access violation.

Use this second view to create the data that you eventually want to make visible to the client. Note that we have two views to the same data: A no-access view that the client knows about and a read-write view that only you know about.

When you're happy with the page of data, you can unmap the second view since you don't need it any more.

Use `VirtualProtect` to change the page protection of the client-visible page to `PAGE_READONLY`. Do this only for the one page that you prepared. This "opens up" that page in the view, converting it from `PAGE_NOACCESS` to `PAGE_READONLY`.

Similarly, when you encounter a write access violation on a page in the client-visible view, you mark the page as dirty and upgrade the page to `PAGE_READWRITE`. When the client closes the database, you unswizzle the dirty pages and write them back out. (If you want to be super-clever, you could also unswizzle the pages and write them out even before the client closes the database. Remember to make the pages read-only, so that you can detect when the client dirties the pages again.)

Notice that the client-visible file mapping now contains a mix of no-access pages, read-only pages, and read-write pages.

There are some obvious optimizations you can perform here.

First of all, you don't have to create a single file mapping for everything. Creating the file mapping will take a commit charge for the entire size of the mapping, even if you end up not using all of it. Instead, you can start with a small file mapping (say, one megabyte), and when you use up all those pages, you create a new file mapping to hold the next megabyte. This creates extra bookkeeping for your page management code, but you won't have more than a megabyte of "extra" memory committed.

Another optimization is to cache the views that you use to prepare the swizzled pages. At one extreme, you could just map them in as read-write and just leave them mapped indefinitely. Or you could keep the few most recent views around, hoping for data locality.

Anyway, that’s the sketch of how you can have a process-wide block of user-mode-managed addresses where you control what happens the first time the client reads from or writes to that page.

Raymond Chen

Follow