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?

devblogs.microsoft.com/oldnewthing/20180125-00

January 25, 2018

Raymond Chen

Last time, we looked at how you can reserve a range of address space and receive notifications when the program first reads or writes a page in the range, in the case where you merely want the notification, or maybe just want to commit blank pages. But what if you want to create nonzero memory instead of just committing blank pages?

For example, you might want to simulate a memory-mapped file, except that instead of being backed by a file, it's backed by some algorithmically generated data. For example, you might be doing pointer swizzling, wherein a large database is incrementally loaded into memory as pages of it are faulted in. As each page faults in, each pointer on the page is updated (swizzled) to point to an as-yet unused page in the reserved region. When an access violation occurs in a reserved page, the code looks up which database page that reserved page corresponds to, loads the page from the database, and then updates each pointer on *that* page to point to an as-yet unused page. From the program's point of view, the database is being paged in on demand.

Pointer swizzling is particularly handy when accessing a very large database on a 32-bit system, because you don't have to memory-map the entire database. The memory usage is the number of pages actually faulted in, and the address space usage is the number of pages referenced by faulted-in pages.

You would handle an access violation on a reserved page by allocating a page of data at the desired location, reading the raw data from the database, and swizzling the pointers. You then mark the page read-only and restart the faulting instruction. ("Look again, and you might find a surprise!")

If you take a write protection violation, then you mark the page as dirty in your data structures, remove write protection from the page, and restart the faulting instruction. When the database file is closed, you unswizzle all the pointers in the dirty pages and write them back to the database.

As with the case discussed last time, you can choose between a structured exception handler (if you need this only for the duration of a function call) or a vectored exception handler (which remains active until explicitly removed). In the case of a swizzled database, you probably would install a vectored exception handler when the database is opened and remove it when the database is closed.

And as with the case discussed last time, you have to watch out for passing these buffers directly to kernel mode, because kernel mode will reject them as invalid. You'll have to turn them from pretend memory to real memory before using them as the source or destination buffer of a kernel mode function.

The nastier problem is multithreading. If one thread chases a swizzled pointer to a reserved page, your code will start filling the page with data. During that time, another thread can chase the same pointer, or another swizzled pointer to the same page, and start accessing the memory while the first thread is still getting the memory ready.

We'll take up this topic next time.

Raymond Chen

Follow